

JavaScript

для тупых

Д.В. Соорег

2020

JavaScript для тупых

Главная

JavaScript для тупых

Maintained? yes

Open Source 

say thanks

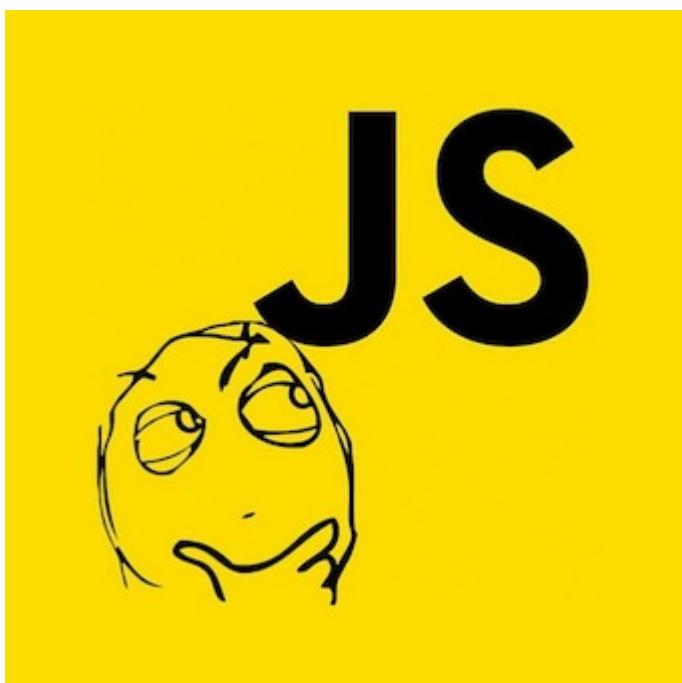
POWERED BY

ELECTRICITY 

Автор: D.B. Cooper

Канал в Telegram: [JavaScript для тупых](#)

Правка материала и вёрстка учебника: [hellosophie](#)



Содержание

1. Формат обучения
2. Ввод в JavaScript. Что это? И для чего нужен?
3. Братки JavaScript'a
4. Переменные в JavaScript
5. Переменные `const`, `let`, `var`
6. Преобразование типов

7. Операторы сравнения
8. Типы данных. **Number**
9. Типы данных. **String**
10. Типы данных. **Boolean**
11. Типы данных. **Null и undefined**
12. Типы данных. Оператор **typeof**
13. Условные операторы: **if** и **?**
14. Циклы: **while**, **for**
15. Операторы
16. Логические операторы
17. Знакомство с **switch**
18. Функции
19. Функции. Возврат значения
20. Стрелочные функции
21. Знакомство с **DOM**
22. **DOM**-элементы. Получение объектов
23. **DOM**-элементы. События. Ввод
24. **DOM**-элементы. События. Часть 2
25. Объекты. Начало игры
26. Массивы
27. Методы массивов: **splice**, **slice**, **concat**
28. Методы массивов: **find**, **findIndex**, **filter**
29. Методы массивов: **map**, **sort**
30. Методы массивов: **reverse**, **split**, **join**
31. Методы массивов: **indexOf**, **lastIndexOf**, **include**
32. Методы массивов: **reduce**, **spread**
33. Методы массивов: **Array.isArray**, **some**, **every**
34. Методы массивов: **flat**, **fill**
35. Планирование вызова функции: **setTimeout**
36. Планирование вызова функции: **setInterval**
37. Объекты. Свойства
38. Объекты. Методы
39. **this**, **call**, **apply**, **bind**
40. Замыкания
41. Прототипы
42. **Promises** (Обещания)

Об учебнике

JavaScript для тупых

Данный учебник подойдет для начинающих web-разработчиков, желающих прокачать свои навыки в JavaScript.

Учебник распространяется **бесплатно**.

За **полное или частичное копирование материалов без согласования с автором** вам будет **стыдно**.

Формат обучения

JavaScript для тупых

Материал будет подаваться жестко, понятно? Знаешь почему? Потому что ты пришёл на курс "JavaScript для тупых" , следовательно, обычная подача материала тебе не подходит по одной из **ниже приведенных причин**:

1. ты реально тупой, как угол в 179° , что не вывозишь то, что вывозит большинство адекватных людей;
2. для тебя обычная подача скучна и от этого материал не усваивается, либо усваивается на весьма короткий срок;
3. при обычной подаче не понимаешь, что происходит, и, что от тебя требуется и зачем;
4. тебе просто-напросто лень читать огромные мануалы.

Короче, если ты в первых трех причинах нашел себя, то радуйся, со мной у тебя есть шансы познать JavaScript .

Если ты попал под четвертую причину – у меня для тебя плохие новости. Ты настолько туп, и даже не понимаешь, что это нужно **в первую очередь тебе самому**, а не кому-то еще. Так что давай, не валяй Ваньку, и бери свою голову с полки, возвращай на плечи и перестань быть ленивцем.

Чем же моя подача лучше обычной? А тем, что твой мозг будет пребывать в шоковом состоянии от того, насколько по-другому ему подают новую информацию.

Если ты не знал, то все, что отличается от **стандартных, обыденных и общепринятых** вещей активизирует и стимулирует мозг, а следовательно, твои извилины начнут шевелиться и появятся первые результаты, которые несомненно всех нас порадуют.

Как будет проходить обучение?

На каждом занятии будет разбор **определенной темы**. В материале будет приведен кусок кода, который я буду разбирать по полочкам и таким языком, чтобы до тебя

наконец-то дошло. Также будет ссылка на **онлайн-редактор** кода (не всегда, но будет), чтобы ты мог поковыряться в коде.

Будут ли домашние задания? Да, но проверять я их не буду и даже не упрашивай. Не забывай – это **нужно тебе**, а не кому-то еще.

Довольно слов, переходим к обучению **JavaScript**.

Ввод в JavaScript. Что это? И для чего нужен?

JavaScript для тупых

Немного истории

JavaScript (JS) язык, который появился в далеком 1995 году, поэтому быть может это не ты не знаешь `JavaScript`, а `JavaScript` не знает тебя. Много рассказывать об истории языка не буду, поскольку в данном курсе это лишняя информация.

Два важных замечания:

1. Существует язык программирования `Java` и существует язык программирования `JavaScript` – так вот, это не один и тот же язык, а совершенно разные языки, которые чаще применяются в разных областях, поэтому не будь Алешкой, не путай языки;
2. `JavaScript` произносится как `"ДжаваСкрипт"`, а не `"ЯваСкрипт"` или как-то по-другому.

Где используется

На самом деле, область применения достаточно широкая. `JavaScript` может использоваться для написания **десктопных** (для компьютеров) **приложений**, которые для пользователя никак не будут отличаться от других программ, написанных, например, на `C++`, `C#` или упомянутом выше `Java`.

Но, все знают (и ты сейчас узнаешь), что чаще всего язык используется в **Web-разработке**. Все эти ваши интернеты насквозь прошиты `JavaScript`-ом и никто от него не сможет спрятаться, если хочет видеть красивый и функциональный сайт.

Если сказать одним предложением – **JavaScript помогает пользователю взаимодействовать с сайтом**. Любой клик мышкой, выделение текста, нажатие клавиши, прокрутка страницы колесиком, свайпы (в случае с мобильными

устройствами) и куча всего другого, что связано с взаимодействием пользователя с сайтом – за всем этим стоит `JavaScript`.

Пример работы `JavaScript`

Ты зашел на страницу Вконтакте, дабы посмотреть на альбомы своих одноклассниц/одногруппниц/коллег. А задумывался ли ты, как все это осуществляется? Как перед тобой появляется лента фотографий?

Вкратце расскажу, что происходит. Когда ты входишь на сайт, то явно куда-то нажимаешь, чтобы добраться до заветного альбома с фотографиями. Скажем, ты нажимаешь какие-то кнопочки, ссылочки, и в этом тебе помогает твой лучший друг, брат, сват – `JavaScript`, именно он отправляет `запрос на сервер`, в котором говорится, что некий юзер, требует альбом с фотографиями другого юзера. Сервер что-то у себя там покрутит, повернит и выдаст ответ в виде списка этих самых фотографий, а `JavaScript` этот ответ примет и обработает для тебя.

Может создаться иллюзия, что фотографии отобразятся у тебя на экране сами собой (вот прям как в твоем семейном альбоме, где ты в 3 года сидишь в розовых колготках сестры), а вот и нет. Если ресурс, который ты посетил, сделан умелыми руками, то сервер вернет список ссылок на эти фотографии, ну или в крайнем случае просто их названия с расширением.

Вроде этого:

```
['photo-1.png', 'photo-2.png', 'photo-3.png']
```

И здесь у тебя встанет вопрос: А каким же образом я вижу все эти прекрасные фотографии своих одноклассниц? И вот здесь (точнее даже немного раньше) снова вступает в работу `JavaScript`.

Он принимает этот список, а затем используя свои возможности и возможности своих братильников `HTML` (разметка веб-страницы) и `CSS` (оформление веб-страницы), выстраивает перед тобой эти самые фоточки, чтобы порадовать тебя.

При этом, ты же помнишь, как выглядит альбом с фотографиями, например, Вконтакте или в Одноклассниках (капец ты старый, если ты там сидишь)? Там

много миниатюр фотографий, и при клике на одну из них фоточка увеличивается почти на весь экран. Так вот за переключение фотографий тоже отвечает **JavaScript**.

Если дочитав до сего момента ты все еще не понял что собой представляет **JavaScript**, ты реально непробиваемая пробка и место тебе на рынке с семками.

Братки JavaScript-а

JavaScript для тупых

В предыдущем уроке ([класс](#)), я упоминал братьев **JavaScript** – это **HTML** и **CSS**. Попробую коротко донести до тебя основную суть.

HTML

Каждая веб-страница, которую ты видишь – состоит из разметки (эта разметка и называется **HTML (HyperText Markup Language)**). **Разметка** – это фундамент страницы. Выглядит он в базовом виде, примерно так:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>Title Page</title>
5  </head>
6  <body>
7      Здесь располагается все, что должен увидеть пользователь
8  </body>
9  </html>
```

Данный код нужно сохранять в файле формата `.html` (например, `index.html`).

Надеюсь ты видишь, что здесь есть разные слова, заключенные в скобки `<>`. Так вот, это не просто так, из разряда – что хочу, то и пишу, эти штуки называются **тегами**. Они практически все являются **парными**. Это означает, что после того, как ты открыл одну скобку `<открывающий тег>`, ты обязательно должен его закрыть `</закрывающий тег>`, иначе браузер не примет твой код.

Пример парного тега:

```
1  <b>Этот текст будет жирным</b>
2
```

```
3 <u>Этот текст будет подчеркнутым</u>
```

Как видишь, сначала я открываю тег, пишу внутри него, то что хочу, а затем закрываю его. Думаю, ты заметил чем отличается **открывающий тег** от **закрывающегося**. Только одним / . Но запомни, этот слэш очень важный.

Хочу предупредить, что не все теги являются парными, например:

- 1
 – тег переноса строки
- 2 <hr> – горизонтальная линия, которая как бы подчеркивает
- 3 содержимое
- 4 – отображение картинки на странице
- 5 и д.р.

Весь список можно погуглить, если понадобится.

В целом, **HTML** служит для того, чтобы разделить информацию на **логические блоки**, для того, чтобы проще этим было управлять (через тот же **JavaScript**) и оформлять через **CSS** .

CSS

Так называемые **каскадные таблицы стилей (Cascading Style Sheets)**. Они служат для того, чтобы навести марафет на странице. Через них, например, устанавливают *цвет фона страницы, размеры шрифтов, цвет этих же шрифтов, устанавливают позиционирование элементов относительно друг друга, относительно окна и так далее.*

Пример CSS-кода:

```
1 .paragraph-1 {  
2   color: green;  
3   font-size: 16px;  
4   font-weight: bold;  
5 }
```

```
6
7 .paragraph-2 {
8   text-align: center;
9   text-transform: uppercase;
10 }
```

Данный код (чаще всего это называют правилами) применяется к любому элементу (тегу) на странице, у которого будет прописан атрибут `class="paragraph-1"` или `class="paragraph-2"` , и даже можно так `class="paragraph-1 paragraph-2"` (в этом случае к элементу с таким атрибутом применяются и правила `paragraph-1` и правила `paragraph-2`), например:

```
1 <p class="paragraph-1">
2   Данный текст будет зеленым, шрифт будет жирным и размером 16px
3 </p>
4
5 <p class="paragraph-2">
6   Данный текст будет располагаться по центру страницы и будет
7   написан прописными буквами
8 </p>
9
10 <p class="paragraph-1 paragraph-2">
11   Данный текст будет зеленого цвета, шрифт будет жирным, он будет
12   располагаться по центру страницы и написан прописными буквами.
13   Правило на размер шрифта 16px не отработает, потому что у нас
14   есть правило `text-transform: uppercase;` , которое явно
15   указывает на то, чтобы все буквы были заглавными
16 </p>
```

CSS-код подключается к странице двумя способами:

1. Прописываем **CSS** код внутри тега `<head>` до его закрытия (`</head>`), обрамляем сам код в тег `<style>...</style>` ;
2. Подключаем через **внешний файл**.

Первый способ:

Тег `<style>...</style>` **HTML-код:**

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>Title Page</title>
5      <link href="styles.css" rel="stylesheet">
6  <style>
7      .paragraph-1 {
8          color: green;
9          font-size: 16px;
10         font-weight: bold;
11     }
12
13
14     .paragraph-2 {
15         text-align: center;
16         text-transform: uppercase;
17     }
18 </style>
19 </head>
20 <body>
21     <p class="paragraph-1">
22         данный текст будет зеленого цвета, шрифт будет жирным и
23         размером 16px
24     </p>
25     <p class="paragraph-2">
26         данный текст будет располагаться по центру страницы и
27         будет написан прописными буквами
28     </p>
29 </body>
30 </html>

```

Второй способ:

Отдельно создаем файл формата `.css`, запихиваем туда наш **CSS код** и подключаем с помощью одиночного тега `<link>` к странице **HTML**:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <title>Title Page</title>
5      <link href=".styles.css" rel="stylesheet">
6  </head>
7  <body>
8      <p class="paragraph-1">

```

```
9  Данный текст будет зеленого цвета, шрифт будет жирным и
10 разместится в центре страницы
11 </p>
12
13 <p class="paragraph-2">
14  Данный текст будет располагаться по центру страницы и будет
15 написан прописными буквами
16 </p>
17 </body>
18 </html>
```

Т.е., эффект будет одним и тем же, стили применяются и в том и в другом случае. Но преимущество **второго метода** в том, что весь **CSS код** отделен от другого, что дает возможность проще разбираться в самом коде и не **говнокодить**.

Подытожим

Надеюсь с помощью данного материала у меня получилось направить тебя в нужное русло и далее ты сам просмотрешь доп.материал в интернете. Я же оставлю тебе ссылочки на код выше, дабы ты мог посмотреть, как и что работает.

Ссылки:

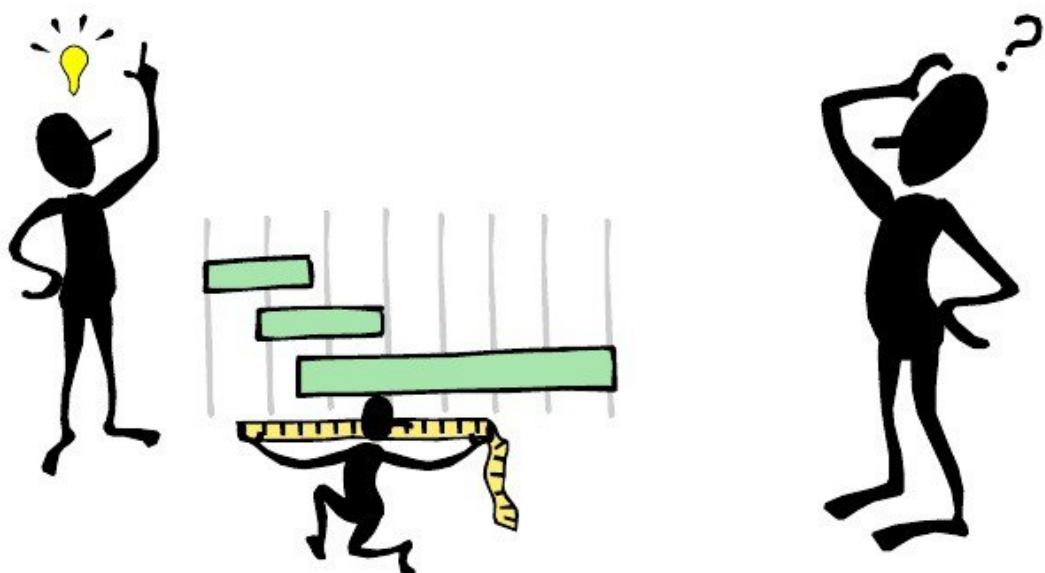
[Learn 1. JavaScriptStupid](#)

[Learn-1.1-JavaScriptStupid](#)

Переменные в JavaScript

JavaScript для тупых

What is a Variable?



Начну с примера: представь, нам нужно сделать простое математическое действие – число 165 умножить (*) на 734 . Затем полученное произведение нужно умножить (*) на 3 , затем поделить (/) на 5 , и каждое полученное значение в результате этих действий нам нужно вывести пользователю.

Самый простой вариант решения задачи:

```
1 console.log(165 * 734);
2 console.log(165 * 734 * 3);
3 console.log((165 * 734) / 5);
```

Фактически, мы сделали то, что требовалось по заданию. Тебя не смущает, что каждый раз мы повторяем одно и то же действие в каждом выражении? Если, не заметил, то мы каждый раз заставляем компьютер вычислять произведение 2-ух чисел . Твоему компьютеру это, конечно же, раз плюнуть, но представь, если там будет другое повторяющееся действие, которое уже будет занимать не 0.0000000000000001 миллисекунды на выполнение, а 3 сек ?

Получится, что выполнение программы будет достаточно долгим процессом. Как же это можно оптимизировать? Как раз в этом нам на помощь приходят **переменные**.

Что же такое переменная? **Переменная** – это выделенная именованная область в памяти для хранения каких-либо данных.

Что это значит? А то, что мы можем создать переменную и записать в нее какое-либо значение, это может быть в виде примитивного значения, в виде строки (`string`), числа (`number`), булева значения (`boolean -> true/false`) или значения, полученного в результате вычисления какого-либо выражения.

Переменная – это подписанная маркером коробка, в которую можно что-то положить, а затем при необходимости взять из нее то, что положил в неё ранее. А если нужно, то даже можно поменять содержимое этой коробки или вообще оставить ее пустой.



В переменную с именем Name мы записываем значение Maria

Пробуем оптимизировать наш код. Получим следующее:

```
1 let productNumbers = 165 * 734;  
2 console.log(productNumbers * 3);  
3 console.log(productNumbers / 5);
```

Как видишь, мы создали переменную с именем `productNumbers` и записали в нее значение произведения. Далее, мы `productNumbers` умножаем на `3`, а следующим действием делим на `5`.

Вся прелесть заключается в том, что значение переменной будет рассчитываться только 1 раз. Т.е., фактически, значением переменной является не само произведение, а результат произведения, т.е.: `121110`. Затем, мы можем обратиться к этой переменной в любой части кода.

Как видишь, код даже немного преобразился, так как стал более читабельным.

Наверное, у тебя возник вопрос, а что это за `let` перед именем переменной? Это **объявление переменной**. Вообще, в JavaScript есть **3 способа объявления переменной**:

```
1 let productNumbers = 165 * 734;  
2 var productNumbers = 165 * 734;  
3 const productNumbers = 165 * 734;
```

Как видишь, их целых 3 брата акробата. Все они делают одно и то же – говорят твоему компьютеру: **разметь в памяти ячейку с именем `productNumbers` и сохрани в неё результат произведения `165 * 734`**.

Чувствуешь, ты уже задаешься вопросом: «зачем их целых 3 и отличаются ли они чем-то?» Да, различаются. Но об этом в следующем уроке.

Переменные. const, let, var

JavaScript для тупых

Итак, приступим к разбору операторов, которые дают жизнь переменным.

Как я и говорил в предыдущем уроке, у нас имеются целых 3 оператора для данного действия. Разберем каждый по порядку.

var

```
var a = 1;
```

Еще мой дед определял переменные с помощью данного оператора, а узнал он это от своего деда и так далее. Короче, все это в прошлом. Это чаще всего можно найти только в проектах с legacy-кодом, т.е., в проектах, которые были давно написаны и их ни разу не рефакторили (*не обновляли код под новые стандарты*). Но, рассказать об этом все равно стоило, так как это основа.

Оператор `var` определяет переменную в текущей области видимости.

Область видимости переменной - это функция, внутри которой она объявлена. Но, если переменную объявили не внутри функции, то ее область видимости - глобальный объект `window`, а это, на секундочку, весь документ.

Переменные объявленные данным оператором, также умеют всплывать, т.е. переменную можно вызвать еще до её определения и это не вызовет никаких ошибок. **Пример:**

```
1 console.log(a); //выведет: undefined
2 var a = 1;
```

Результатом выполнения такого кода будет вывод в консоль `undefined`, а так происходит только в том случае, когда переменная уже определена, но у нее нет никакого значения.

let

Этот оператор очень похож на оператор `var`. Главное их различие - **область видимости** (но об этом мы говорим позже).

Если попробовать повторить тот же самый код:

```
1 console.log(a); //выведет: ReferenceError: a is not defined  
2 let a = 1;
```

Как видишь, на этом моменте возникнет ошибка, в которой **JavaScript** говорит тебе, что ты косорукий и пытаешься использовать топор, когда его еще и не придумали.

Оператор `let` работает более предсказуемо, с ним проще, он твой дружище. **Не используй var - используй let.**

const

```
const a = 1;
```

Оператор очень обидчивый, и по всякой чепухе его лучше не дергать. Переменная объявленная с помощью данного оператора - это коробка, в которую **ОБЯЗАТЕЛЬНО** нужно что-то положить, а если уж положил, то **не трогать!**

На содержимое такой переменной можно только смотреть, любоваться. И даже не вздумай менять содержимое такой переменной. У тебя не получится. От слова «совсем».

У тебя лишь получится словить ошибку вида:

"Uncaught TypeError: Assignment to constant variable" .

Коротко и без аналогий: переменная объявленная таким оператором обязательно должна иметь значение и его нельзя менять.

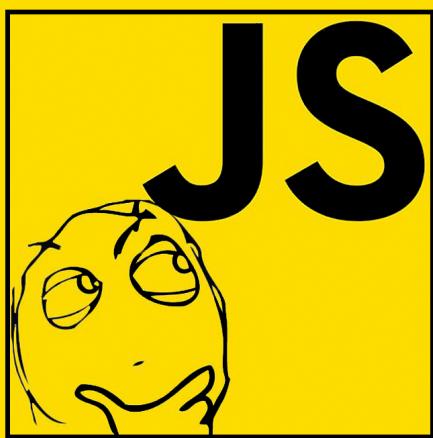
Если ты понял, что ты ничего не понял, то не переживай. Это не ты тупой, и даже не я. Со всеми этими областями видимости, которые ты сейчас вообще не понимаешь мы столкнемся еще ни раз в следующих уроках.

Я сторонник того, чтобы понимание приходило во время практики. Поэтому максимально усвой то, что написано выше, а все что не понятно — поймешь со временем.

Преобразование типов

JavaScript для тупых

Преобразование типов



Очень часто при разработке возникает ситуация, когда нужно преобразовать типы. Ты столкнешься с этим на практике, даже не думай об обратном.

Преобразование к String (строковый тип)

Представим, нам нужно преобразовать переменную `возраст`, которая имеет тип `Number` (числовой) в новую переменную типа `String`.

```
1 let ageNumber = 19;  
2 let ageString = ???;
```

Как же мы будем это делать? Приведу в пример несколько способов:

```
1 let ageNumber = 19;
```

```
2 let ageString = ageNumber + ''; // Вариант 1: так делал мой
3 // прадед
4 let ageString = `${ageNumber}`; // Вариант 2: так уже чуть
5 // лучше, чем делал мой дед
6 let ageString = String(ageNumber); // Вариант 3: вот так гораздо
7 // лучше, если будешь делать так
```

Как видишь, ничего сложного. Сделав одним из способов, ты почти на 100% будешь уверен, что все точно преобразовалось в строку. Почему почти? Потому как такое отлично сработает, если ты работаешь с *примитивными типами данных*: `Number`, `String` (да, сам `String` можно привести в `String`), `Boolean`, `null` и даже `undefined`. Но, такое преобразование не прокатит с `объектами`. Но об объектах мы всецело будем говорить чуть позже, так как еще к ним недостаточно подготовились.

Важно отметить, что при сложении переменных, если хотя бы одна из них имеет строковый тип, может произойти неприятная неожиданность:

```
1 let a = '1';
2 let b = 2;
3 let c = a + b; // получим значение: '12'
```

Оператор `+`, в данном случае может сыграть злую шутку. Он просто-напросто склеит значения переменных в одну строку. И, кстати, неважно в каком порядке мы будем складывать `a + b` или `b + a`.

Преобразование к `Number` (числовой тип)

Преобразовывать к числовому типу немного повеселее, так как в этом случае уже не все так просто и могут кое-где приключиться интересные истории.

Ради примера, представим обратную ситуацию:

```
let ageString = '19';
```

Теперь наша начальная переменная имеет тип `String`.

Приступим к преобразованию:

```
1 let ageString = '19';
2 let ageNumber = Number(ageString); // (1)
3 let ageNumber = + ageString; // (2)
4 let ageNumber = ageString / 1; // (3)
5 let ageNumber = ageString * 1; // (4);
```

Итак, в пример я привел **4 варианта**. Рассмотрим каждый из них:

1. **Первый вариант** называется **явным преобразованием типа**. Функция `Number` попробует преобразовать переданный в нее аргумент в строку. При этом, если вокруг цифр будут стоять пробелы, то они обрежутся и это не будет ошибкой. А вот если в строке будут содержаться какие-либо символы, кроме цифр, то в переменную запишется значение `NaN` (**Not-A-Number**). Если же строка была пустая, то в переменную запишется значение – `0`.
2. **Второй вариант** очень часто встречается в коде разработчиков. Это является неким **лайфхаком**. В данном случае, оператор `+` полностью заменяет вызов функции `Number`, т.к. их поведение полностью совпадает.
3. **Третий и четвертый варианты – как делать можно, но не нужно.** Примеры привел для того, чтобы показать, что `умножение` И `деление` так же заставляет переменные приходить к типу `Number`.

Есть некоторые особенности, которые важно запомнить при вызове функции

`Number`:

```
1 Number(undefined); // получим значение: NaN
2 Number(null); // получим значение: 0
3 Number(true); // получим значение: 1
4 Number(false); // получим значение: 0
```

Преобразование к `Boolean` (логический тип)

Преобразование к логическому значению по сравнению с первыми двумя, пожалуй, самая простая операция.

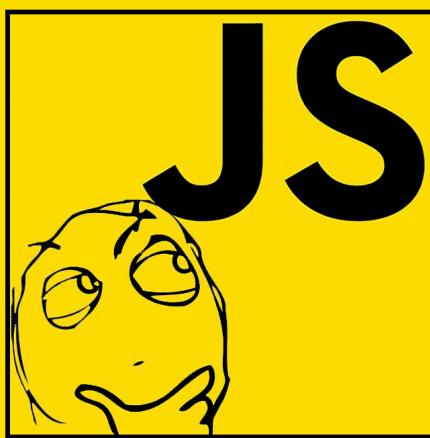
```
1 Boolean(1); // получим значение: true
2 Boolean('строка'); // получим значение: true
3 Boolean(0); // получим значение: false
4 Boolean(null); // получим значение: false
5 Boolean(undefined); // получим значение: false
6 Boolean(NaN); // получим значение:false
7 Boolean(''); // получим значение: false
```

Если коротко, то все, что имеет какое-то значение, после преобразования будет иметь значение `true`, а все остальное – `false`.

Операторы сравнения

JavaScript для тупых

Операторы сравнения



Сегодня подробнее хочу рассказать об **операторах сравнения** и где они используются.

Надеюсь, услышав фразу **операторы сравнения** ты вспомнил хотя бы школьную математику. Если вспомнил, то ты еще не потерян, а вот если нет, придется тебя возвращать к жизни =).

Операторы сравнения в **JavaScript** ровно такие же, как и в математике, ничего особо нового. Но, я обязан рассказать. Начнём.

Список всех операторов сравнения в JavaScript:

- **больше меньше:** `a > b` , `b < a`
- **больше или равно, меньше или равно:** `a >= b` , `b <= a`
- **равно:** `a == b`
- **не равно:** `a != b`

Здесь хочу обратить твоё внимание на операторы **равно** `=` и **не равно** `!=`

Дело в том, что нужно четко понимать, что при проверке на равенство ты всегда должен использовать, **как минимум два знака равно** (`==`), а не один. Это нужно делать по одной причине – если ты напишешь один знак (`a = b`), то **JavaScript** рассмотрит эту операцию, как *присваивание значения переменной `b` в переменную `a`, а не как сравнение*. **Это важно и нужно запомнить раз и навсегда!**

А также то, что оператор **не равно** пишется именно со знаком восклицания (`!=`), и никак иначе.

Если ты внимательно читаешь, то заметил, что я написал **как минимум два знака равно**. Наверное, у тебя назрел вопрос, а сколько же их может быть еще? Об этом позже, потерпи =)

Результат сравнения

Как и операторы **сложения/вычитания**, **умножения/деления**, которые, как я думаю, не нуждаются в объяснении, операторы сравнения так же возвращают результат. Тип этого значения – **Boolean** (логический).

Пример:

```
1 4 > 1; // (1) – получим логическое значение true
2 3 < 1; // (2) – получим логическое значение false
3 3 <= 4; // (3) – получим логическое значение true
```

Все до предела просто. В примере (1), значение `4`, действительно, больше чем значение `1`, следовательно, сравнение истинно и является правдой (`true`).

Напомню, что **правда/истина** в логическом типе равна значению `true`.

В примере (2), мы получаем логическое значение `false`, поскольку это сравнение **неверно/ложно**. Значение `3`, на самом деле, больше, чем значение `1`.

Пример (3) уже немного интереснее. Здесь уже проверяются 2 условия. Если хотя бы одно из условий **истинно**, то мы всегда получаем логическое значение `true`. И здесь получается, что хоть значение `3` и не равно значению `4`, но оно меньше, а значит одно из условий было выполнено.

Стоит понимать, что чаще всего приходится сравнивать не только типы `Number` (числа), но и `String` (строки), `Boolean` (логический). Короче говоря, сравнивают все, что не лень =)

```
1 'строка' == 'строка'; // получим значение: true
2 true == false; // получим значение: false
```

Также строки можно проверять на `>` или `<`. В таком случае, строки будут проверяться **посимвольно**. Но, так как на практике это применяется крайне редко, я просто приведу самый простой пример:

```
1 'а' < 'я'; // получим значение: true
2 (т.к., 'я' // в алфавите идет после 'а')
```

На самом деле, в данном случае проверка идет не по алфавиту, а по таблице `Unicode`, которую использует **JavaScript**, но, в данный момент я не хочу углубляться в эту тему, по той причине, что на практике это практически никогда не используется.

До этого мы сравнивали только **одинаковые типы**, но что будет, если попытаться сравнить **разные типы**?

Сравнение разных типов

Ничего сложного в сравнении разных типов нет. **JavaScript** будет стараться приводить каждое из сравниваемых значений к числу:

```
1 '09' == 9; // получим значение: true ('09' преобразуется
2 // в значение 9 типа Number)
3 '100' < 4; // получим значение: false ('100' преобразуется
4 // в значение 100 типа Number)
5 true == 0; // получим значение: false (true преобразуется
6 // в значение 1 типа Number)
7 false == 0; // получим значение: true (false преобразуется
8 // в значение 0 типа Number)
```

Естественно, если постараться сравнить это:

```
'строка' > 3;
```

JavaScript будет пытаться преобразовать значение 'строка' в число, а так как это невозможно – мы получим значение **NaN (Not-A-Number)** и это выражение всегда будет **ложным** и возвращать **логическое** **false**

Строгое сравнение

А теперь вернусь к вопросу, который я задал в самом начале, а именно: **Сколько же знаков равно может использоваться при сравнении?**

Ответ: 2 или 3.

В чем отличие между 2-я и 3-я знаками?

Когда мы сравниваем значения с помощью `==`, то **JavaScript** пытается привести сравниваемые значения к типу **Number (числу)**, что может привести к неожиданным результатам, если не разбираться в этой теме. Например:

```
1 '' == false; // (1) получим значение: true
2 0 == false; // (2) получим значение: true
3 1 == true; // (3) получим значение: true
```

В случае (1) , пустая строка приводится к **Number** и преобразуется в числовое значение **0** , и значение **false** преобразуется в числовое значение **0** . Поэтому получается, что выражение **истинно**.

В случае (2) , значение **false** приводится к числовому значению **0** и выражение также становится **истинным**.

В случае (3) , значение **true** приводится к числовому значению **1** и выражение тоже оказывается **верным**.

Вроде бы во всех трёх случаях все логично, **НО**, если подумать, то как в таком случае отличить, например, значение **0** от значения **false** ?

А вот здесь как раз к нам на помощь приходит **строгое сравнение**: **==** .

При **строгом сравнении** не происходит никаких приведений типов. Пример:

```
1 0 === ''; // получим значение: false
2 1 === true; // получим значение: false
3 1 === '1'; // получим значение: false
4 '' === false; // получим значение: false
5 true === true; // получим значение: true
6 1 === 1; // получим значение: true
```

В данном случае, **JavaScript** сначала проверяет типы сравниваемых значений. Если они не совпадают, то **JavaScript** прекращает проверку и выдает результат: **false**

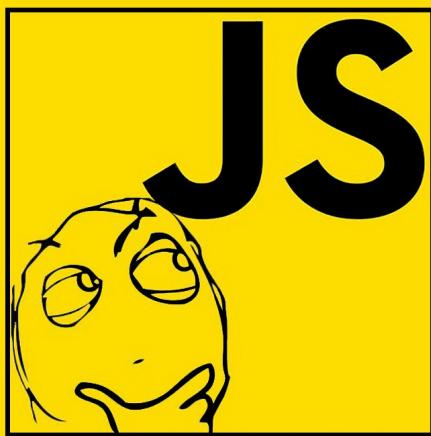
В целом, по операторам сравнения на этом думаю закончить, так как мы рассмотрели все важные моменты.

Данный урок очень важен для понимания. Внимательно перечитай несколько раз те места, которые не до конца понял.

Типы данных. Number

JavaScript для тупых

Типы данных Числовой(Number)



JavaScript использует так называемую динамическую типизацию . Это означает, что, всецело, типы данных в языке есть, но переменные не привязываются к ним.

Кроме тех типов данных, имеющихся практически в любом языке программирования (ЯП), JavaScript имеет несколько типов, которыми он решил ни с кем не делиться, но о них чуть позже.

Числовой тип (Number)

В числовой тип входят целочисленные значения и значения с плавающей точкой. Это несколько не стандартно. Обычно, в других ЯП, целочисленные значения и значения с плавающей точкой разделяются на 2 типа: `integer` и `float` , соответственно.

Примеры переменных типа number:

```
1 let a = 1;  
2 let b = 1.5;
```

Переменные типа `number` мы можем спокойно складывать, делить, вычитать, умножать и т.д (производить любые математические действия).

Цихверки и всё такое – это понятно. Но есть еще, так называемые **специальные числовые значения**.

В их список входят: `Nan`, `Infinity`, `-Infinity`.

NaN

NaN - Not-A-Number. По-русски – не число. Немножко логика поломалась, сейчас, да?) Числовое значение, которое не число. Так мог завернуть только наш любимый **JavaScript**. Но, ты это, привыкай. С **JavaScript** вообще легко не будет. Он будет крутить твой мозг как пропеллер.

Итак, `Nan`, фактически, означает **вычислительную ошибку**. Например, ты взял и решил умножить строковую переменную на число, ну, или поделить и засунуть результат в переменную. Именно в этот момент и вступает в бой тип `Nan` и назначается этой переменной.

Пример возникновения `NaN`:

```
let a = "John" * 20; // выведет NaN
```

Infinity

Помнишь, как тебя в школе учили, что на ноль делить нельзя? А может быть, ты даже в университете уже отучился, где тебе уже сказали, что на самом деле это делать можно. Только вот в этом случае получится бесконечность, что в языке **JavaScript** является значением `Infinity`.

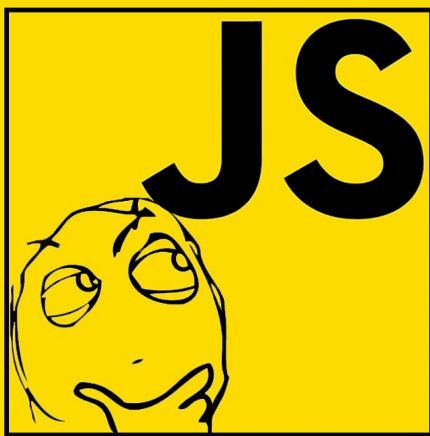
Наравне с физической величиной `c` (скорость света), `Infinity` в расчетах ведёт себя примерно так же. Если `Infinity` суммировать с любым числом, то значение будет все равно `Infinity`.

Так же, хотелось бы отметить, что при математических действиях не может случится такого момента, что скрипт намертво упадет и перестанет выполняться. Самый страшный случай, который может произойти в расчетах – это переменная может превратиться в `Nan`, там где рассчитывалось увидеть число. Поэтому, будьте бдительны в работе с числами и учитесь их правильно обрабатывать, в особенности, когда эти данные приходят от пользователя.

Типы данных. String

JavaScript для тупых

Типы данных Строка(String)



Строка (String)

Строка в **JavaScript** должна заключаться в кавычки. Можно в **двойные**, можно в **одинарные**. **Правильный вопрос: как НУЖНО?** А, нужно, заключать строку в **одинарные кавычки**. Это правило хорошего тона среди разработчиков. Ни в каких спецификациях это не указано, **JavaScript** будет работать и с **двойными** и с **одинарными**, но разработчики между собой покумекали и решили, что в **одинарных** смотрится лучше. Так что, будем тоже использовать **одинарные**, всегда и везде.

Получается, что можно делать так:

```
1 "значение переменной"  
2 'значение переменной'
```

Так же строку можно заключить в обратные кавычки:

`значение переменной`

Обратные кавычки появились в ES6 и добавили массу удобных штук в JavaScript.

Они как бы расширяют возможности обычной строки (в одинарных и двойных кавычках). Внутрь такой строки можно положить значение переменной или какое-то выражение и оно станет частью этой строки.

Например:

```
1 let name = 'John';
2 console.log(`Hello, ${name}`);
3 // выведет в консоль: Hello, John
```

Как видишь, в строке с обратными кавычками использован синтаксис `${...}`, где вместо `...` можно подставить любую переменную или даже вызвать функцию и полученное значение вклейится в строку. Такой синтаксис будет работать только со строкой, которая обрамлена обратными кавычками.

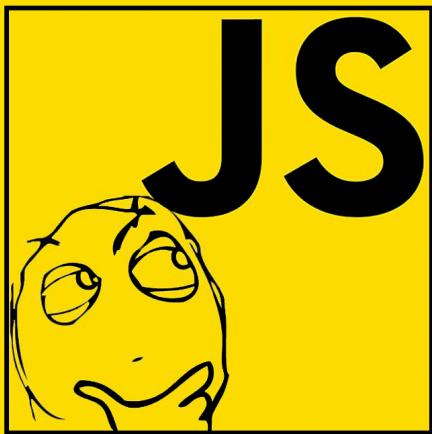
Итог

Скорее всего некоторые вещи могут звучать не очень понятно. Но, запомни, в обучении, и, тем более, в теории много непонятного. И, зачастую, даже опытные проггеры не знают некоторых вещей или не до конца понимают, как они работают. А ты стоишь у самых истоков, поэтому ничего страшного. Позже ты во всем разберешься =).

Типы данных. Boolean

JavaScript для тупых

Типы данных Логический(Boolean)



Логический (`boolean`) тип, несмотря на свою простоту, является одним из самых важных в любом ЯП. Кстати, некоторые называют этот тип булевым, поэтому не пугайтесь, если где-то наткнетесь на такое название.

`Boolean` состоит всего из двух значений: `true` (истина) и `false` (ложь).

Из этого вытекает, что данный тип используют для хранения значений **да или нет**.

Стоит отметить, что `true` и `false` не нужно заключать в кавычки! Иначе это приведет к ошибке типов. Стоит только заключить логические значения в кавычки, как они сразу же превращаются в тип `String`.

К примеру, у нас в коде имеется переменная, на значение которой может повлиять пользователь. Пользователь перед регистрацией может принять или отклонить соглашение нашего сайта. Т.е., у пользователя есть возможность отметить мышкой соглашение. В зависимости от этого наша переменная будет менять свое значение.

```
1 let agreementAccepted = false; // значение по умолчанию
2 agreementAccepted = true; // если пользователь поставил галочку
3 // и принял соглашение
```

Так же, у пользователя мы можем запросить ввести свой возраст, чтобы понять, можно допускать пользователя к информации на нашем сайте или нет.

Например, на своем сайте ты рассказываешь о хирургических операциях, с подробными фотографиями. В таком случае, ты должен будешь закрыть эту информацию от людей, которые **младше 18 лет**.

```
1 let age = 17;
2 let allowAccess = age >= 18;
```

Представь, что в переменной `age` содержится тот возраст, который указал пользователь на сайте. В переменную `allowAccess`, мы записываем значение логического выражения: `17 >= 18`. В данном случае, значение будет равно `false`.

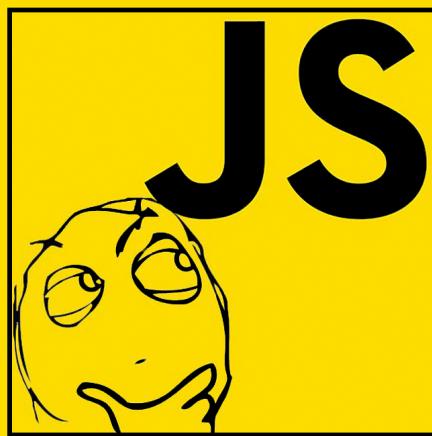
Таким образом, проверив значение переменной `allowAccess`, мы можем понять, можно показывать данные пользователю или нет. Для этого в ЯП существуют такие конструкции, как `условия`.

Вообще, чаще всего логический тип служит для дальнейшей проверки чего-либо, т.е, этот тип очень сильно связан с условиями. Но тема `условий` будет отдельным уроком.

Типы данных. null и undefined

JavaScript для тупых

Типы данных null и undefined



Тип null

Данный тип формируется только из одного единственного значения – `null`. Да-да, `null` является и типом и значением одновременно. Используется он для того, чтобы пометить что-либо, как ничего, пустышка и т.п.

Зачастую `null` используется при инициализации какой-либо переменной, которая изначально не имеет никакого значения.

Например:

```
let agreementAccepted = null;
```

Т.е, этим мы инициализировали переменную с начальным значением `null`. Это часто применяется и об этом ты узнаешь дальше, когда мы будем заниматься практикой. Если не указать значение переменной при её инициализации, то значение переменной будет равно `undefined`.

Тип `undefined`

Тип `undefined`, так же как и `null` является и типом и значением. `undefined` в дословном переводе означает **не определено**. Если переводить с точки зрения ЯП, то `undefined` нужно понимать как **значение не присвоено**.

К примеру:

```
let agreementAccepted;
```

В данном случае, значение переменной `agreementAccepted` будет равно `undefined`. Логически может казаться, что `null` и `undefined` являются одним и тем же и разницы между ними нет. Но, не ведись на эту логику – она неверная.

В мире профессиональной разработки нужно четко понимать разницу между этими типами.

Запомни раз и навсегда! Использовать `undefined` уместно только в тех местах кода, где ты хочешь проверить существует ли значение как таковое или была ли чем-то инициализирована переменная.

Если хочешь инициализировать переменную с пустым значением, то используй только `null`. Никаких осознанных переменных с присвоенным в неё значением `undefined` в коде быть не должно!

Мы хотим инициализировать переменную возраста с пустым значением:

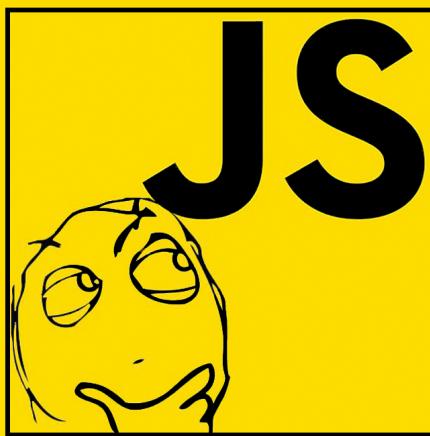
```
1 let age; // не надо так  
2 let age = undefined; // не надо так
```

```
3 let age = null; // только так!
```

Типы данных. Оператор typeof

JavaScript для тупых

Типы данных Оператор typeof



Сейчас рассмотрим с тобой такую ситуацию: ты писал код, написал тысячу миллионов строк, у тебя 100500 переменных и ты уже не помнишь, какой тип данных лежит в конкретной переменной, что делать, как узнать?

Оператор typeof

Этот оператор пояснит тип переменной. Сразу же к использованию:

```
1 typeof 'оператор typeof'; // результат: string
2 typeof 100500; // результат: number
3 typeof undefined; // результат: undefined
4 typeof true; // результат: boolean
5 typeof null; // результат: object (??? что? как это возможно?)
```

Как видим, оператор почти всегда говорит правду. Но, вот с типом `null` происходит заминка. Просто нужно запомнить, что это **известная ошибка**, которую

не могут поправить из-за сохранения совместимости и всякое такое, ну, короче, просто запомни, что `typeof` никогда не расскажет тебе правды о `null`.

Кстати, оператор можно вызывать **2-я способами**:

```
1 typeof('оператор'); // (1)
2 typeof 'оператор'; // (2)
```

Оба случая и (1) и (2) имеют жизнь, как использовать - решать только тебе, но я бы рекомендовал вызывать его как **стандартную функцию**, т.е. как показано в случае (1).

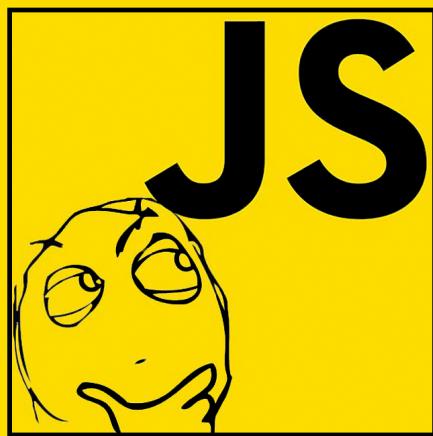
Практическое применение оператора

Так как в **JavaScript НЕТ строгой типизации**, но зачастую нужно проверить тип переменной для совершения каких-либо операций, приходится использовать данный вид оператора. Так что, это твой один из первых полезных братанов в **JavaScript**. Запомни его и то, что он иногда говорит не правду =)

Условные операторы: if и ?

JavaScript для тупых

Условные операторы: if и '?'



@javascriptstupid

Условные операторы нужны для того, чтобы выполнять какие-либо действия в зависимости от условий.

Для этого в **JavaScript** существует два оператора: `if` и `?` (да-да именно вопросительный знак).

Оператор `if` (если)

Сразу же к примеру:

```
1 let age = prompt('Какой сейчас век?', '');
2
3 if (age == 21) {
4     alert('Верно, сейчас XXI век');
5 }
```

Поясню за `prompt`. Это функция, которую предоставляет **JavaScript** для того, чтобы взаимодействовать с пользователем.

При запуске этого кода, у тебя на экране появится окно с вопросом:

`Какой сейчас век?`. И ниже, в этом же окне будет доступно поле для ввода ответа.

Какой сейчас век?



Пример работы функции `prompt`

Введенный тобой ответ, автоматически запишется в переменную `age`, после нажатия кнопки `ок`. Кстати, чтобы ты не ввел в это поле: **строку, число**, да что угодно – `prompt` приведет это к типу `String`. Т.е, переменная `age` будет иметь тип `String`.

Дальше по коду у нас идет условный оператор `if`. Вся эта конструкция, на самом деле, очень проста. Два три упражнения и ты навсегда запомнишь, как работает этот оператор. Итак, разберем подробно:

```
1 if (age == 21) {  
2     alert('Верно, сейчас XXI век');  
3 }
```

Ты можешь видеть, что после оператора `if` идет выражение в скобках – это обязательное условие, чтобы оператор начал работать. В скобках мы указываем то выражение, которое хотим проверить на истинность (**живой пример**).

После скобок оператор `if` будет ожидать действие, которое нужно выполнить в том случае, если выражение в скобках будет истинным (читать про [преобразование типов](#)), т.е. выдаст нам логическое `true`. Здесь есть **2 варианта** как написать это действие:

```
1 // первый вариант:  
2 if (age == 21) alert('Верно, сейчас XXI век');  
3  
4 // второй вариант:  
5 if (age == 21) {  
6     alert('Верно, сейчас XXI век');  
7 }
```

Разница между этими двумя – **наличие или отсутствие фигурных скобок**.

Первый вариант подразумевает, что при выполнении условия ты хочешь, чтобы выполнилась только одна команда. Любая последующая операция, в таком случае, будет выполняться всегда, вне зависимости от условия. Пример:

```
1 if (age == 21) alert('Верно, сейчас XXI век');  
2 alert('Я буду появляться вне зависимости от \  
3 выполнения/невыполнения условия');
```

В данном случае, вторая функция `alert` будет выполняться всегда, вне зависимости от того, выполнилось условие `age == 21` или нет ([живой пример](#)).

Второй же вариант подразумевает то, что ты хочешь выполнить несколько операций при выполнении условия. Все эти операции ты должен поместить внутрь фигурных скобок:

```
1 if (age == 21) {  
2     alert('Верно, сейчас XXI век');  
3     alert('Я буду появляться только если выполнится условие');  
4 }
```

Обе команды сработают только в том случае, когда условие будет выполнено.

*Всегда и везде, рекомендую тебе использовать фигурные скобки, даже, если ты хочешь выполнить только одну команду. Использовать фигурные скобки считается лучшей практикой (**best practice**).*

У оператора `if` имеется младший брат – оператор `else`.

Оператор `else` (иначе)

Этот оператор не может жить без своего старшего брата `if`, поэтому он используется всегда только вместе с ним. Вызвать `else` отдельно невозможно, да и вызов этот не несет в себе никакого смысла и логики.

Оператор `else` организует блок, внутрь которого можно засунуть те команды, которые должны выполниться в том случае, если условие, которое указано в скобках после оператора `if` не выполнилось. Не отходя от кассы, пример:

```
1 if (age == 21) {  
2     alert('Верно, сейчас XXI век');  
3 } else {  
4     alert('Неверно! Ты ответил неправильно!');  
5 }
```

Получается, что если мы ответим неправильно и условие не выполнится, т.е. вернет нам логическое `false`, то отработает блок `else` и у тебя на экране появится окно с сообщением: "Неверно! Ты ответил неправильно!" ([живой пример](#)).

Еще один момент – иногда нужно проверить несколько условий сразу. Для этого можно использовать блок `else if`. Пример:

```
1 if (age == 21) {  
2     alert('Верно, сейчас XXI век');  
3 } else if (age < 21) {  
4     alert('Ты указал век меньше, чем текущий');  
5 } else if (age > 21) {
```

```
6     alert('Ты указал век больше, чем текущий');
7 } else {
8     alert('Скорее всего, ты ввел не число');
9 }
```

Когда используется блок `else if` все условия проверяются поочередно друг за другом сверху вниз. Если какое-либо условие выполняется, то все дальнейшие условия игнорируются ([живой пример](#)).

Оператор '?

Этот оператор называют **тернарным** (состоящим из трёх частей). Это единственный оператор, который имеет **3 операнда**.

Зачем нам нужен этот оператор? Представь ситуацию, тебе нужно записать значение в переменную, но это значение зависит от условий.

Используя условный оператор `if` мы можем это сделать так:

```
1 let temperature = prompt('Сколько градусов за окном?', '');
2 let weather = null;
3
4 if (temperature >= 0) {
5     weather = 'больше 0';
6 } else {
7     weather = 'меньше 0';
8 }
9 alert('Температура сегодня: ' + weather);
```

ЕСЛИ (`if`) температура больше, либо равна нулю, то в переменную `weather` мы записываем значение `true`, ИНАЧЕ (`else`) записываем в нее значение `false`.

Не кажется ли тебе, что достаточно много кода ради такого простого действия?!

Потому и был придуман **тернарный оператор?** .

Ведь если использовать его, получится намного короче:

```
1 let temperature = prompt('Сколько градусов за окном?', '');
2 let weather = (temperature >= 0) ? 'больше или равна 0' : \
3   'меньше 0';
4
5 alert('Температура сегодня: ' + weather);
```

Первым операндом оператора `?` является `условие`, которое мы проверяем. Затем ставится сам оператор `?`, после него пишем то, что хотим присвоить в переменную `weather` в случае выполнения условия, а после `:` пишем то, что хотим присвоить в случае, если условие не выполнилось.

Разберем подробнее эту строку:

```
let weather = (temperature >= 0) ? 'больше 0' : 'меньше 0';
```

Следует читать и понимать эту строку так: **ЕСЛИ** `temperature >= 0`, **ТО** в переменную `weather` мы записываем значение `'больше 0'`, **ИНАЧЕ** записываем значение `'меньше 0'` ([живой пример](#)).

Я выделил жирным **3 момента**. При разборе таких выражений читай их именно так. Перед условием проговаривай слово **ЕСЛИ**, тернарный оператор `?` заменяй на слово **ТО**, а знак `:` заменяй на слово **ИНАЧЕ**. Если запомнишь, то будешь без труда понимать данную конструкцию.

Согласись, так код выглядит намного лучше.

Проверять несколько условий сразу можно не только с помощью оператора `if`, но и при помощи **тернарного оператора** `?`. Усовершенствуем пример выше:

```
1 let temperature = prompt('Сколько градусов за окном?', '');
2
3 let weather = (temperature > 0) ? 'больше 0' :
4   (temperature == 0) ? 'равна 0' :
5     (temperature < 0) ? 'меньше 0' :
```

```
6      'Ты точно ввел только цифры? 🤔';
7
8 alert('Температура сегодня: ' + weather);
```

Наверное, выглядит немного страшно ([живой пример](#)). Но, поверь, здесь опять нет ничего страшного. Хотя, признаюсь, когда я только начинал постигать **JavaScript**, то эта конструкция вселяла в меня ужас, так как мне казалось, что в этом очень просто запутаться. Но, как оказалось, все намного проще.

Здесь мы использовали **тернарный оператор** аж целых 3 раза. Сейчас объясню, как это работает с множественными условиями.

```
1 let weather = (temperature > 0) ? 'больше 0' :
2   (temperature == 0) ? 'равна 0' :
3   (temperature < 0) ? 'меньше 0' :
4   'Ты точно ввел только цифры? 🤔';
```

Данным кодом мы говорим **JavaScript-у**: если переменная (`temperature > 0`) , то присвой в переменную `weather` значение `'больше 0'` . Если это не так, то проверь условие `(temperature == 0)` и если оно **истинно**, то присвой в переменную `weather` значение `'равна 0'` . Если и это условие не проходит, то мы просим **JavaScript** проверить следующее условие `(temperature < 0)` , и в случае его выполнения присвоить значение `'меньше 0'` . Ну, а если все условия проверены, и ни одно не оказалось **истинным**, мы просим, чтобы **JavaScript** записал в переменную `weather` значение `'Ты точно ввел только цифры? 🤔'` .

Думаю, что на сегодня достаточно. В нескольких местах оставил ссылки на куски кода в онлайн-редакторе и подписал их «живой пример». Ты можешь перейти по ссылкам, посмотреть код и запустить его нажав кнопку `Run`

The screenshot shows the JSFiddle interface. At the top, there's a navigation bar with icons for cloud, run, save, fork, set as base, tidy, collaborate, embed, update (green button), settings, and a power icon. Below the navigation is a sidebar on the left with sections for Author (showing a profile picture and the handle javascriptstupid), Fiddle meta, Resources (URL, cdnjs), Async requests, and Other (links, license). A MongoDB Atlas advertisement is present. The main area has tabs for HTML (selected), CSS, and JavaScript + No-Library (pure JS). A red arrow points from the 'Run' button in the top bar down to the 'Run' button in the center of the JSFiddle interface. In the center, there's a message in Russian: 'Для выполнения кода нажми эту кнопку' (Press this button to execute the code). Below it is the code editor with the following JavaScript:

```
1 let temperature = prompt('Сколько градусов за окном?', '');
2
3 let weather = (temperature > 0) ? 'больше 0' :
4     (temperature == 0) ? 'равно 0' :
5     (temperature < 0) ? 'меньше 0' :
6     'Ты точно ввел только цифры? =)';
7
8 alert('Температура сегодня: ' + weather);
```

онлайн-редактор JSFiddle

Домашнее задание

1. Что выведется на экран, если выполнить данный код:

```
1 if ('') {
2     alert('1');
3 }
```

2. Используя `if .. else`, напиши код, который задаст пользователю вопрос (с помощью `prompt()`) и примет от него ответ. В зависимости от ответа, выведи на экран с помощью `alert()`: если пользователю **больше или равно 18**, то выведи: «**вам разрешено посещение сайта**»; если пользователю **меньше 18**, то выведи: «**вам запрещено посещение сайта**»;

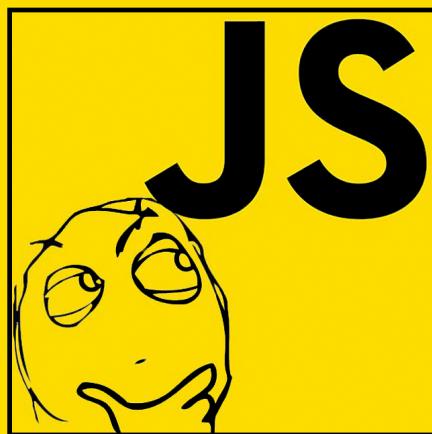
3. Самостоятельно потренируйся с **тернарным оператором** `?`

Выполненным домашним заданием вы можете поделиться в комментариях к этому уроку.

Циклы: while, for

JavaScript для тупых

Циклы while, for



@javascriptstupid

Цикл – это такая штука, которая позволяет выполнять какие-либо операции много раз.

Например, с помощью функции `alert` нам нужно вывести на экран числа от `1` до `3`. Как бы ты сейчас это сделал? Если ты не знаешь о том, как работают циклы, ты напишешь примерно так:

```
1 alert(1);
2 alert(2);
3 alert(3);
```

Возможно, в каких-то ситуациях можно допустить такой код – но это **нежелательно** и считается **плохим тоном**. И, наконец, представь, что задача изменилась и нужно вывести числа от `1` до `1000` – ты же не хочешь писать 1000 строк, верно? Для этого и придумали `циклы`.

Цикл while

Цикл имеет следующий синтаксис:

```
1 while (условие) {  
2     // команды которые нужно выполнить  
3 }
```

Все команды внутри цикла `while` выполняются до тех пор, пока условие истинно (`true`) .

И сразу же переделаем нашу задачу под цикл ([живой пример](#)):

```
1 let i = 1;  
2 while (i <= 3) {  
3     alert(i);  
4     i++;  
5 }
```

Что же здесь происходит?! Пойдем по строкам:

`let i = 1;` – определяем переменную `i` со значением `1` ;

`while (i <= 3) {` – запускаем цикл `while` и устанавливаем условие, при выполнении которого цикл будет продолжать работу. Так как нам нужно вывести значения от `1` до `3` и это означает, что цикл должен выполняться до тех пор, пока значение переменной `i` не станет равным значению `3` . Т.е., значение должны быть **меньше, либо равны** `3` . А для того, чтобы значение `i` изменялось, после каждого выполнения цикла, мы должны **увеличивать значение переменной `i` на единицу** (это происходит на строке `4`);

`alert(i);` - выводим текущее значение переменной `i` ;

`i++;` – увеличение значения переменной `i` на единицу. Оператор `++` называется инкрементом . Его смысл заключается в том, чтобы увеличивать значение на единицу. Делаем мы это для того, чтобы следующий проход цикла проверил новое условие.

Цикл do...while

Если цикл `while` сначала проверяет условие и только при его истинности выполняет команды, то цикл `do...while` – сначала выполняет все команды и только потом начинает проверять условие.

Вот его синтаксис:

```
1 let i = 1;
2 do {
3     alert(i);
4     i++;
5 } while (i <= 3);
```

Разница очень маленькая и надеюсь ты ее уловил ([живой пример](#)).

Цикл for

Еще один важный цикл, который используется куда чаще – это цикл `for` .

Синтаксис:

```
1 for (let i = 1; i <= 3; i++) {
2     alert(i);
3 }
```

Этот цикл тоже не является сложным. После оператора `for` , в скобках указываются последовательно: `начало` , `условие` , и `шаг` ([живой пример](#)).

Первое значение, `let i = 1;` означает, что мы определили переменную `i` со значением `1`. Это начальное значение переменной, с которым выполнится первый оборот цикла.

Вторым значением, в скобках, является **условие**, при выполнении которого выполнится тело цикла, а именно функция `alert(i);`

Третьим значением является **указание шага**, т.е., насколько нужно увеличить значение `i` за один оборот цикла. Т.к. увеличить нам нужно на единицу, то снова пользуемся инкрементом `++`.

Существуют еще некоторые особенности использования циклов, но с ними мы будем знакомиться дальше, на практике. Пока же, мы только узнаем о циклах, поэтому сначала нужно привыкнуть к их синтаксису.

Подробнее об операторах инкремента/декремента расскажу в следующем уроке.

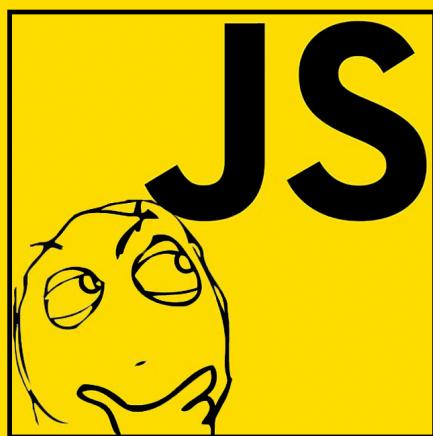
Домашнее задание

1. Напиши код, который выводит на экран с помощью цикла `while` значения от `50` до `100`.
2. Напиши код, который выводит на экран с помощью цикла `do...while` значения от `20` до `30`;
3. Напиши код, который выводит на экран с помощью цикла `for` значения от `200` до `250`;

Операторы

JavaScript для тупых

Операторы



@javascriptstupid

Большинство операторов известно нам еще с начальных классов, к примеру:

- +
- -
- /
- *
- =

Они все используются в **JavaScript**, но есть нюансы.

Нюансы касаются операторов: +, -, *. Они в **JavaScript** умеют немного больше, чем в обычной математике.

Унарный/бинарный плюс

К примеру, с помощью оператора `+` можно складывать не только **числа**, но и **строки**:

```
1 'a' + 'b'; // получим: 'ab'  
2 '1' + '3'; // получим: '13'
```

Стоит быть крайне внимательными, когда пытаетесь сложить 2 числа, а точнее, это вы думаете, что там 2 числа, а на самом деле там могут оказаться строки. И всё, приплыли, выдаст ошибку.

Кстати, значения, к которым ты хочешь применить любой оператор, называются **аргументами** или **операндами**.

Чтобы такой чумы не случилось, можно использовать функцию **JavaScript** под названием **Number**, а можно использовать еще одну супер способность оператора `+`: **преобразование к числу**. Это я тоже уже показывал в одном из предыдущих уроков:

```
1 let a = '45';  
2 let b = +a;
```

В данном случае, переменная `b` будет иметь значение 45 и тип, внимание – **Number**. Оператор `+` поставленный перед любым значением, будет пытаться преобразовать это значение к типу **Number**. В данном случае его называют **унарным плюсом**, что означает лишь одно: **плюс применяется только к одному операнду**.

Запись такого типа, не должна тебя пугать:

```
console.log(+ '1' + + '3'); // получим значение: 4
```

Здесь происходит следующее: тот плюс, который стоит непосредственно перед складываемым значением является **унарным**, он имеет больший приоритет для **JavaScript**. Т.е., сначала произойдет преобразование к числам, и только потом отработает **бинарный плюс**.

Бинарный плюс – оператор, который применяется уже к 2-м operandам (аргументам) .

Небольшой пример бинарного плюса:

```
1 + 4;
```

В предыдущем уроке о **циклах**, я упомянул такие термины, как **инкремент** и **декремент**. Разберем эти понятия.

Инкремент

Инкремент просто-напросто **увеличивает** текущее значение переменной **на единицу**:

```
1 let a = 3;
2 a++;
3 console.log(a); // получим значение: 4
```

Как видишь, плюсики стоят после переменной. Такая запись инкремента называется **постфиксной формой**. Но есть еще **префиксная форма**.

Префиксная форма точно так же, как и постфиксная **увеличивает значение на единицу**. Но в чем же тогда разница, спросишь ты? А вот в чем:

```
1 let a = 5;
2 console.log(a++); // что выведет?
```

Думаешь, что в консоль выводится значение `6`? А вот и нет, в консоль выводится старое значение `5`. **Значение** в данном случае **увеличивается после** того, как ты вызвал инкремент, а не в момент его вызова.

А вот **префиксная** работает так, как ты ожидал от **постфиксной**:

```
1 let a = 5;
2 console.log(++a); // выведет значение 6
```

Префиксная форма меняет значение переменной перед тем, как вывести в консоль и поэтому все работает более логично.

Декремент

Декремент уменьшает текущее значение переменной на единицу:

```
1 let a = 3;
2 a--;
3 console.log(a); // получим значение: 2
```

Декремент тоже имеет **префиксную** и **постфиксную** формы.

Важное замечание: инкремент, как и декремент можно использовать только с переменной. Просто к значению его применить нельзя – словите ошибку.

Возведение в степень

JavaScript позволяет достаточно просто возвести любое число в любую степень. Для этого используется оператор `**`:

```
1 2 ** 3; // читать, как "два в третьей степени": 8
2 3 ** 2; // читать, как "три во второй степени": 9
```

Остаток от деления

И еще один полезный оператор – `%`. Оператор позволяет нам **получить остаток после деления**. Пример:

```
1 11 % 2; // получим значение: 1
2 8 % 3; // получим значение: 2
```

В первом случае: `11` не делится на `2` без остатка. Максимальное число, которое делится на `2` без остатка в данном примере – это `10`. Получается, что остаток здесь равен единице (`11 - 10`).

Во втором случае все ровно так же: Максимальное число, которое делится без остатка в данном случае – это `6`. Вычитая из значения `8` значение `6`, получаем: `2`.

Именно значение остатка и предоставляет нам этот оператор.

Домашнее задание

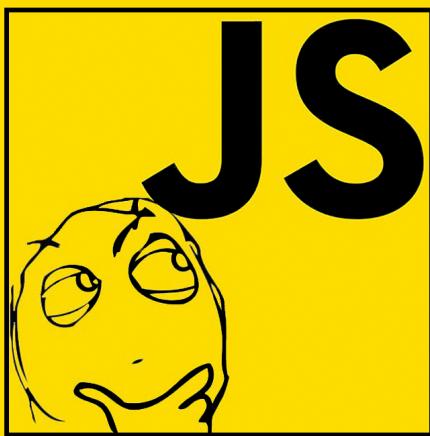
Сегодня совсем байтовое домашнее задание. Весь код я вынес в новый для тебя онлайн-редактор (*Repl.it*). Ещё раз изучи код и попробуй со всем поиграться самостоятельно.

[Ссылка на редактор с кодом](#)

Логические операторы

JavaScript для тупых

Логические операторы



@javascriptstupid

Тема тоже очень важная и одна из **фундаментальных**.

Логические операторы

Существует всего **три логических оператора**.

Логические операторы: `||` , `&&` , `!`

Идём по порядку.

Оператор `||` (ИЛИ)

Данный оператор представляет собой две палки ~~колбасы~~, и, если, его называть по-русски, читая код, то он читается как **ИЛИ**. Он используется всегда с 2-я operandами (аргументами), т.е. является **бинарным**.

Результат работы этого оператора – логическое значение `true` или `false`.

Если хотя бы один из операндов (аргументов) дает логическое `true`, то и всё выражение в целом будет возвращать `истину`. Пример:

```
1 let a = false;
2 let b = true;
3
4 if (a || b) {
5   alert('Я буду появляться всегда, т.к. один из аргументов \
6   возвращает логическое значение: true ');
7 } else {
8   alert('Я никогда не появлюсь в данном случае');
9 }
```

Из примера понятно, что, если один из аргументов возвращает `true`, то все выражение в условии возвращает `истину`. Еще один пример:

```
1 let a = 1;
2 let b = 0;
3
4 if (a || b) {
5   alert('Я буду появляться всегда, т.к. один из аргументов \
6   возвращает логическое значение: true ');
7 } else {
8   alert('Я никогда не появлюсь в данном случае');
9 }
```

Здесь все отработает ровно так же. После приведения типов, переменная `a` выдаст значение `true`, а переменная `b` вернет значение `false`, но т.к., выполнилось правило и один из аргументов возвращает `true`, то это означает, что условие будет выполнено.

Чтобы условие с оператором `||` не выполнилось нужно, чтобы оба операнда (аргумента) возвращали `false`.

Более или менее жизненный пример. Представь, у тебя есть форум, на котором у каждого пользователя есть возможность прикреплять файлы к сообщению. Но эта возможность доступна не каждому пользователю, а только при выполнении

определенных условий: пользователю не меньше 21 года или он написал более 500 сообщений.

Код:

```
1 let ageUser = 18;
2 let countMsgs = 800;
3
4 if ((ageUser >= 21) || (countMsgs > 500)) {
5     alert('Вам разрешено прикреплять файлы к сообщению');
6 } else {
7     alert('Вам запрещено прикреплять файлы к сообщению');
8 }
```

Несмотря на то, что пользователю меньше 21 года условие выполнится, потому что он написал более 500 сообщений .

Еще пример:

```
1 let ageUser = 18;
2 let countMsgs = 120;
3
4 if ((ageUser >= 21) || (countMsgs > 500)) {
5     alert('Вам разрешено прикреплять файлы к сообщению');
6 } else {
7     alert('Вам запрещено прикреплять файлы к сообщению');
8 }
```

В этом случае условие не выполнится, потому что ни одно из условий внутри if не вернуло логическое значение true .

Оператор && (И)

Этот оператор более придирчивый чем || . Если оператору ИЛИ нужно хотя бы одно истинное значение, то оператор И не потерпит никаких false – всё должно

быть `true` и только так. Только когда все операнды возвращают значение `true`, оператор `&&` вернет `true`.

Пример:

```
1 alert(true && false); // вернет: false
2 alert(false && false); // вернет: false
3 alert(false && true); // вернет: false
4 alert(true && true); // вернет: true
```

Возьмем тот же пример с форумом и переделаем под оператор **И**. Получается, что условия задачи меняются: чтобы пользователь мог прикреплять файлы к сообщению ему должно быть `больше 21` года и он должен написать `более 500 сообщений`.

```
1 let ageUser = 18;
2 let countMsgs = 600;
3
4 if ((ageUser >= 21) && (countMsgs > 500)) {
5     alert('Вам разрешено прикреплять файлы к сообщению');
6 } else {
7     alert('Вам запрещено прикреплять файлы к сообщению');
8 }
```

Здесь ты потерпишь фиаско, т.к., пользователю `менее 21` года. Дальше **JavaScript** даже второе условие проверять не будет, потому как первое уже **запустилось** и вернуло `false`.

Оператор ! (НЕ)

Данный оператор не похож на первые два, но также очень важен.

Оператор является **унарным**, т.е., применяется он только к одному операнду (аргументу). Ставится оператор всегда перед аргументом.

Все что делает оператор можно описать в **2-х шагах**:

1. Приводит операнд (аргумент) к логическому значению (`true/false`);
2. Возвращает логическое значение противоположное тому, что получил в шаге 1

Сразу же простой пример:

```
1 alert(!true); // вернет: false
2 alert(!false); // вернет: true
3 alert(!'string'); // вернет: false
4 alert(!0); // вернет: true
5 alert(!8); // вернет: false
```

Надеюсь все понятно. Поэтому сразу же перейду ко второй способности этого оператора. Можно взять и поставить два знака `!!` перед аргументом. Что произойдет в этом случае? Не читай ответ, а подумай, основываясь на том, что ты уже прочитал об операторе. Итак, барабанная дробь! А произойдет следующее:

1. Первый знак `!` приведет аргумент к логическому значению;
2. Затем вернет противоположное логическое значение полученное в шаге 1 и на этом полномочия первого оператора **НЕ** завершатся;
3. Второй оператор **НЕ** проделает те же 2 шага и вторым шагом снова приведет к значению (вернув его).

Да что за...? Да-да, здесь все просто на самом деле. Данный метод используют очень часто при **преобразовании в логический тип**, вместо функции `Boolean(..)`. Сам подумай, насколько проще написать так:

```
1 let str = 'string';
2 let a = !!str;
```

Нежели так:

```
1 let str = 'string';
```

```
2 let a = Boolean(str);
```

В первом случае запись короче, а смысл абсолютно не меняется.

Домашнее задание

Задача:

Пользователю с помощью функции `prompt` предлагается ввести `возраст`.

Если возраст `больше 18` лет, то с помощью еще одного `prompt`, нужно запросить имя пользователя.

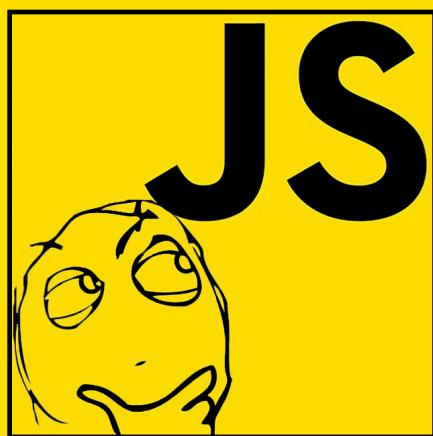
Все эти данные нужно сохранить в `переменных`.

Как только ты получишь от пользователя все необходимые данные, ты должен сделать следующие проверки: **если** пользователю `больше 25` лет и его имя `John`, ты должен с помощью функции `alert()` вывести фразу `"Welcome, John"`. В противном случае ты должен вывести ему фразу `"Who are you? I don't know you"`.

Знакомство с switch

JavaScript для тупых

Знакомство с switch



@javascriptstupid

Сегодня расскажу о продвинутом операторе `if`, а именно о конструкции `switch`.

`if` и `switch` очень близки по своей сути, однако, в множестве случаев спора, кто лучше: тот или другой, победителем выйдет именно `switch`.

Теперь подробнее об этой конструкции. `switch`, фактически заменяет в коде несколько конструкций `if`, что, порой, позволяет избегать лишних проверок (соответственно, существенно сократить код).

Вся конструкция `switch` выглядит следующим образом:

```
1 switch(arg) {  
2     case 1:  
3         // действия  
4     break;  
5     case 2:  
6 }
```

```
7      // действия
8  break;
9
10 default:
11     // действия
12 }
```

Теперь сразу же разберем на примере:

```
1 let arg = 5;
2
3 switch(arg) {
4     case 1:
5         alert('Значение переменной arg равно 1');
6         break;
7
8     case 3:
9         alert('Значение переменной arg равно 3');
10    break;
11
12    case 5:
13        alert('Значение переменной arg равно 5');
14        break;
15
16    default:
17        alert('Значение переменной arg равно: ' + arg);
18 }
```

Разберем код ([живой пример](#)).

Мы определили переменную `arg` и записали в неё значение `5`.

Затем, эту переменную `arg` мы как бы “скормливаем” конструкции `switch`.

Конструкция берет значение переменной, которую ей скормили, и проверяет не совпадает ли это значение с одним из значений указанных в блоках `case`. Если значение совпадает с каким-либо значением указанным после ключевого слова `case`, то все действия выполняются, которые указаны внутри этого блока `case` до оператора `break`. Оператор `break` говорит **JavaScript-у**, что нужно выйти из конструкции `switch`.

Т.к., значение переменной `arg` равно `5`, то выполнится код, который написан внутри последнего блока `case`, т.е. этот:

```
alert('Значение переменной arg равно 5');
```

Блок `default`, находящийся внизу конструкции `switch` является необязательным. Он служит для того, чтобы код внутри него выполнился в том случае, если значение переменной скормленной в `switch` не совпало ни с одним из значений, указанных в блоках `case`.

Т.е, если сейчас взять и поменять значение переменной `arg`, например, на `6` (в примере выше), то отработает блок `default`.

Важные заметки:

`switch` очень любит точность от программиста. Поэтому, если ты пишешь так:

```
1 ...
2 case 3:
3 ...
```

Это означает, что ты ожидаешь от переменной именно числового значения `3`, а не строки `'3'`. `switch` просто не отработает так, как ты от него того ожидаешь, если ты не будешь думать о типах данных при сравнении.

Второй момент: кроме того, что блок `default` является необязательным, в конце этого блока можно не писать оператор `break`, т.к. этот код находится в конце конструкции `switch`, а значит `JavaScript` так или иначе выйдет из этой конструкции.

Домашнее задание

Написать конструкцию `switch`, которая будет проверять сколько опыта получил ваш герой в игре, в зависимости от количества убитых врагов.

Условия задачи:

- За 1 врага ему начисляется 100 очков опыта
- За 3 врагов ему начисляется 400 очков
- За 5 врагов начисляется 700 очков
- За 10 врагов начисляется 1000 очков

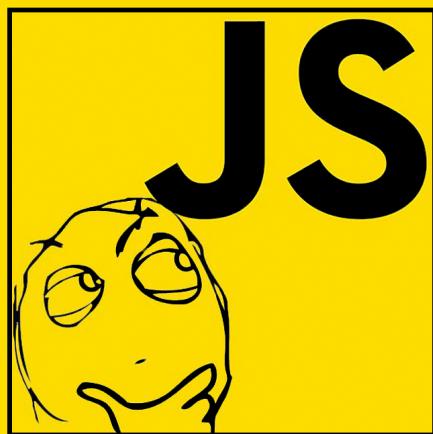
Если же он набрал другое количество очков, то умножай количество убитых врагов на 100 очков.

Далее, запроси (с помощью функции `prompt`), сколько игрок убил врагов, а потом с помощью `alert` выведите игроку информацию о том, сколько очков опыта он заработал.

ФУНКЦИИ

JavaScript для тупых

ФУНКЦИИ



@javascriptstupid

За все предыдущие уроки, ты уже не раз пользовался различными функциями, такими как: `alert()` , `prompt()` .

Сегодня же наступил день, когда ты сможешь написать свою собственную **функцию** и вызывать ее в любой части кода.

Функции придумали для того, чтобы не плодить одинаковый код в разных частях проекта. При разработке существует огромное множество мест, где это может произойти. Но, повторять код – большая ошибка и проблема. Не делай так.

Представь, у тебя есть форум. У каждого пользователя есть свой профиль, где он имеет возможность пополнить свой внутренний счет. На форуме очень много разделов и кнопка «**Пополнить счет**» встречается в нескольких местах сайта. Как обрабатывать множество кнопок? Что ж, мы можем для каждой кнопки написать один и тот же код обработки, но тогда это будет копированием одного и того же кода через весь проект. Т.е, мы можем делать примерно так:

```
1 let amount = +prompt('На какую сумму Вы хотите пополнить \
2 счет?');
3 alert(`Счет пополнен на сумму ${amount} $`);
```

И мы можем копировать этот код по всему проекту, где нужно будет обработать нажатие кнопки. Но представь, что потом тебе нужно будет поменять валюту пополнения, т.е. сменить значок `$` на значок рубля (`₽`). Тебе придется искать по всему проекту, куда ты копировал эти обработчики и в каждом куске заменять этот знак.

Не легче ли написать код один раз и затем просто вызывать его там, где это требуется? Для этого и придумали функции.

```
1 function callPayment() {
2     let amount = +prompt('На какую сумму Вы хотите пополнить\
3     счет?');
4     alert(`Счет пополнен на сумму ${amount} $`);
5 }
```

Немного разберем синтаксис ([живой пример](#)). Для того чтобы написать свою функцию, сначала нужно написать ключевое слово `function`, после него написать **имя будущей функции**, после имени, обязательно, открыть и закрыть скобочки `()`, затем открыть **тело функции** открывающей фигурной скобкой `{` написать весь требуемый код и закрыть тело функции закрывающей фигурной скобкой `}`. *Весь этот процесс называется определением функции.*

Теперь, когда тебе потребуется выполнить код, который находится внутри этой функции ты просто должен вызвать эту функцию. Вызов функции выглядит так:

```
callPayment();
```

Вот и все. В этом и есть прелесть функций. Если вызвать функцию 3-5-10 раз, то весь код выполнится ровно столько же раз ([живой пример](#)) :

```
1 callPayment();
2 callPayment();
3 callPayment();
```

И, теперь, если нужно будет что-то поменять в процессе пополнения счета, то все что тебе будет нужно – это изменить саму функцию в одном месте.

Аргументы функции

Возьмем ту же самую функцию:

```
1 function callPayment() {
2     let amount = +prompt('На какую сумму Вы хотите пополнить\
3     счет?');
4     alert(`Счет пополнен на сумму ${amount} $`);
5 }
```

После имени функции есть скобки – () . Они не просто для красоты, а еще и для того, чтобы засунуть в функцию какие-либо данные. Сейчас все поймешь.

Представь, у пользователя где-то в настройках профиля указана валюта, в которой он хочет пополнять счет. Следственно, значок валюты в этой строке должен измениться на ту, которую выбрал пользователь в настройках своего профиля.

```
alert(`Счет пополнен на сумму ${amount} $`);
```

И еще встает один вопрос: для каждой валюты писать свою функцию? Ответ: нет.

Все, что нужно, это передать внутрь функции валюту указанную в профиле пользователя. Сделать это очень просто. Изменим нашу функцию:

```
1 function callPayment(currency) {  
2     let amount = +prompt('На какую сумму Вы хотите пополнить\  
3     счет?');  
4     alert(`Счет пополнен на сумму ${amount} ${currency}`);  
5 }
```

Как видишь, в скобках появилась переменная с именем `currency`. Этую переменную можно использовать внутри функции. Для полноты картины ([живой пример](#)):

```
1 function callPayment(currency) {  
2     let amount = +prompt('На какую сумму Вы хотите пополнить\  
3     счет?');  
4     alert(`Счет пополнен на сумму ${amount} ${currency}`);  
5 }  
6  
7 let userCurrencyRub = '₽'; // валюта: рубль  
8 let userCurrencyUSD = '$'; // валюта: доллар  
9  
10 callPayment(userCurrencyRub); // вызываем функцию с \  
11 // валютой: рубль  
12 callPayment(userCurrencyUSD); // вызываем функцию с \  
13 // валютой: доллар
```

Если не понял, поясняю: мы берем нужную нам валюту и засовываем ([передаем](#)) ее внутрь **функции**. Переданное нами значение `userCurrency` записывается в переменную `currency`, которую мы написали при определении функции внутри `()`. Затем, полученную переменную `currency`, можно использовать внутри функции и в ней будет лежать именно то значение, которое мы передали.

Аргументы позволяют сделать любую функцию более гибкой.

Значение аргументов по умолчанию

```
1 function callPayment(currency) {  
2     let amount = +prompt('На какую сумму Вы хотите пополнить\  
3     счет?');  
4     alert(`Счет пополнен на сумму ${amount} ${currency}`);  
5 }
```

```
6  
7 callPayment();
```

Если вызвать функцию без передачи в нее аргумента `currency`, то значение этого аргумента (переменной) внутри функции будет равно `undefined`.

Чтобы не допустить этого, можно установить значение аргумента по умолчанию. Для этого внесем небольшие изменения в определении функции ([живой пример](#)):

```
1 function callPayment(currency = '$') {  
2     let amount = +prompt('На какую сумму Вы хотите пополнить\\'  
3     счет?');  
4     alert(`Счет пополнен на сумму ${amount} ${currency}`);  
5 }  
6  
7 callPayment();
```

Все что мы изменили, это приравняли аргумент `currency` к значению `$`. В таком случае, если вызвать функцию и не передать в нее значение аргумента – по умолчанию будет равен уже не `undefined`, а `$`.

При этом, если мы все-таки передадим какое-то значение, то **JavaScript** как бы забудет про то, что мы назначали значение по умолчанию и возьмет именно то значение, которое мы передали.

Думаю, на этом знакомство с функциями закончим. Пока вам необходимо будет разобраться с определением функции и ее аргументами.

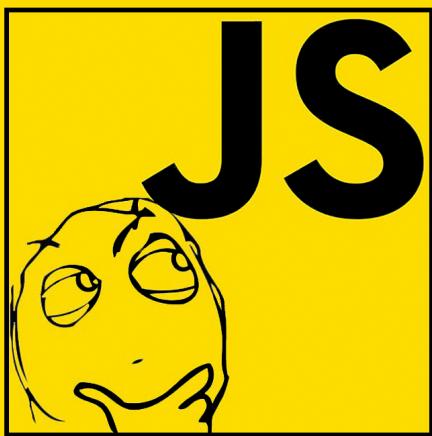
Домашнее задание

1. Определить функцию, которая будет запрашивать число (с помощью `prompt`), умножать его на `327` и делить на `10`. И с помощью `alert` выдавать полученное значение;
2. Определить функцию с аргументом, которая будет запрашивать число (с помощью `prompt`), умножать его на то значение, указанное в качестве аргумента. И с помощью `alert` выдавать полученное значение.

Функции. Возврат значения

JavaScript для тупых

Функции Возврат значений



@javascriptstupid

Я упустил очень важный момент в предыдущем уроке про **функции** в **JavaScript**. А именно, я забыл рассказать тебе, что функция умеет **возвращать значения**.

Представь, тебе нужно написать функцию, которая будет что-то считать и то, что она посчитает ты должен записать в переменную.

К примеру, напишем функцию, которая будет возводить число в квадрат и возвращать это значение. Начну поэтапно. Сначала напишем функцию, которая просто высчитает квадрат:

```
1 function sqr(a) {  
2     let result = a * a;  
3 }  
4  
5 sqr(4);
```

Мы вызвали функцию `sqr` со значением `4`. Внутри функции, мы считаем квадрат числа и записываем его в переменную `result`. И здесь встает вопрос: как же вернуть это значение, чтобы использовать его где-то ниже по коду? Ведь в данном случае функция сделает вычисления и никто об их результате не узнает.

Чтобы научить функцию возвращать значения, используется ключевое слово `return`. После этого слова мы должны указать, что нужно вернуть, а вернуть мы должны значение переменной `result`. Допилим нашу функцию и получим следующее:

```
1 function sqr(a) {  
2     let result = a * a;  
3  
4     return result;  
5 }  
6  
7 sqr(4);
```

Теперь функция возвращает значение и нам осталось только записать его. А куда мы записываем все значения? Правильно, в `переменные`. Допиливаем код и получаем:

```
1 function sqr(a) {  
2     let result = a * a;  
3  
4     return result;  
5 }  
6  
7 let result = sqr(4); // теперь в переменной будет храниться\  
8 // значение 16  
9 alert(result);
```

Ну и как бы все ([живой пример](#)). Мы записали в переменную `result` тот результат, который возвратила нам функция. В данном случае – это значение `16`. И ниже для примера с помощью `alert()` выводим полученное значение.

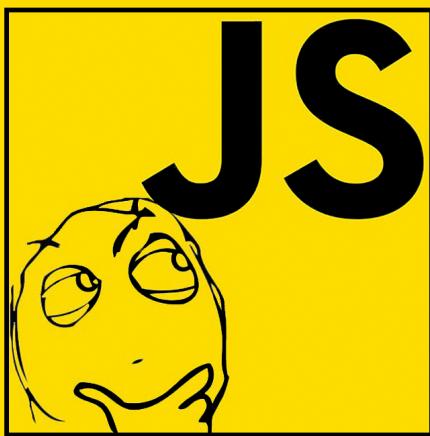
Домашнее задание

Написать функцию, которая с помощью `prompt` запрашивает число у пользователя. Затем эта функция умножает введенное число на `100` и возвращает это значение. Результат функции должен быть записан в `переменную`.

Стрелочные функции

JavaScript для тупых

Стрелочные функции



@javascriptstupid

Данный тип функций пришел из стандарта ES6 и завоевал сердца всех проггеров. А все потому, что зачастую они намного облегчают жизнь, отчасти они более простые и их запись занимает меньше строчек кода.

Синтаксис

Сразу же начну с синтаксиса:

```
1 let func = () => alert('Я стрелочная функция!');  
2  
3 func();
```

Сразу же приведу код для сравнения с обычной функцией:

```
1 function func() {  
2     alert('Я стрелочная функция');  
3 }
```

Две функции, которые выполняют абсолютно одинаковые действия.

В случае со **стрелочной функцией** ты уже не используешь ключевого слова `function`. И как, видишь, функцию ты записываешь в переменную `func`, которую потом вызываешь.

В `func` ты записываешь следующую конструкцию `() => alert(...)`. Эта конструкция самая простая, когда тебе нужно выполнить только **одно действие**. Если же нужно выполнить **несколько действий**, то здесь ты обязан добавить фигурные скобки `{}` и после них поставить символ `;`. Пример:

```
1 let func = () => {  
2     alert('Я стрелочная функция');  
3     alert('Честно');  
4 };
```

Аргументы функции

Если функция должна принимать аргументы, то здесь тоже всё просто. По аналогии с обычными функциями ты можешь указать аргументы в скобках при определении функции и передавать их туда при вызове.

```
1 let func = (a) => {  
2     alert(a);  
3 };  
4  
5 func(3); // выдаст в alert значение 3
```

Возвращение значения в стрелочных функциях

Так же **стрелочные функции** позволяют не писать ключевое слово `return`, если функция что-то должна вернуть. Но работает это только в том случае, если выполняется одно действие.

```
1 let func = () => 1 + 1;
2
3 let func2 = () => {
4     return 1 + 1;
5 };
```

Функции `func` и `func2` – полностью идентичны, поэтому зачем писать `return` и добавлять фигурные скобки, если можно обойтись и без них.

Если в функции нужно выполнить несколько действий и вернуть результат, то здесь уже в любом случае нужно добавлять фигурные скобки и ключевое слово `return`.

```
1 let func = () => {
2     let result = 2 + 2;
3
4     return result;
5 };
```

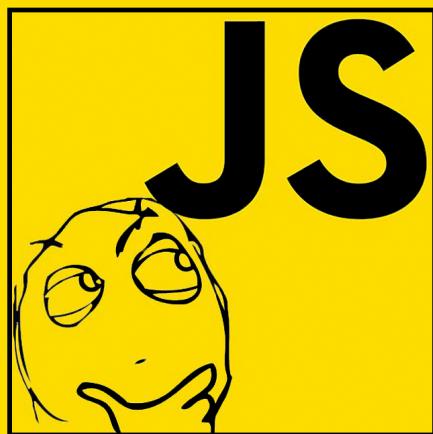
Домашнее задание

Напиши **стрелочную функцию**, которая будет запрашивать с помощью `prompt` два числа у пользователя и `перемножать` их между собой. Полученное значение функция должна возвращать с помощью ключевого слова `return`. Результат выполнения функции сохрани в `переменную`.

Знакомство с DOM

JavaScript для тупых

Знакомство с DOM



@javascriptstupid

Что такое DOM?

Все теги в **HTML**-документе рассматриваются браузером как `объекты`. Внутри тега могут содержаться другие теги. Такие теги будут являться `дочерними`. Проще говоря, весь **HTML**-документ представляет собой `дерево тегов с разными разветвлениями`. В целом, это и есть `DOM (Document Object Model)`.

Итак, самая простая страница:

```
1 <html>
2 <head>
3     <title>Page title</title>
4 </head>
5 <body>
6     <p>Текст</p>
7     <button>Click Me</button>
8 </body>
9 </html>
```

Данный пример – самая простая **HTML**-страница. Напомню, что эту разметку нужно сохранять в формате `.html`, чтобы ее мог открыть браузер.

Если внимательно посмотреть на приведенные в примере теги, то можно заметить, что внутри одних тегов мы чётко видим другие.

Внутри тега `<html>` лежат все остальные теги: `<head>`, `<body>`. Внутри же этих тегов, например, `<head>` лежит тег `<title>`.

Все, что существует внутри **HTML**-документа, является частью **DOM**-дерева.

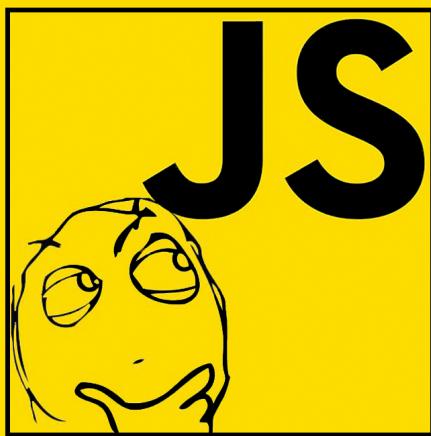
Если честно, то различных аспектов с **DOM** достаточно много, но не хочется на них заострять большое внимание, потому как это будет больше теории, нежели практики. А я пытаюсь научить тебя больше практической части.

Если совсем подытожить, то **DOM** позволяет нам крутить и вертеть всеми элементами нашего **HTML**-документа. **Мы можем изменить положение элемента, цвет, да почти всё что угодно.**

DOM-элементы. Получение объектов

JavaScript для тупых

DOM-элементы Получение объектов



@javascriptstupid

Итак, поговорим об элементах и о том, как начать ими управлять. Сразу код ([живой пример](#)):

```
1 <html>
2 <head>
3     <title>Page Title</title>
4 </head>
5 <body>
6     <p>Текст</p>
7     <button>Click Me</button>
8
9     <script src="main.js"></script>
10 </body>
11 </html>
```

Для примера, давай попробуем средствами **JavaScript** изменить цвет текста, который находится внутри тега `<p>` на зеленый. Звучит довольно просто, но давай

разберемся что нам нужно от **JavaScript**:

1. Выбрать нужный элемент;
2. Изменить цвет текста этого элемента

Итак, приступим к решению 1-й задачи.

Чтобы выбрать конкретный элемент, нам необходимо этот самый элемент пометить таким образом, чтобы мы могли выбрать конкретно его. Другими словами, мы должны назначить управляемому элементу (именно так стоит называть элемент, с которым ты хочешь что-то сделать) **личный идентификатор**. Этот идентификатор должен быть только у этого элемента. Здесь на ум приходит аналогия с номером и серией паспорта. У каждого человека он свой.

Для того, чтобы назначить элементу идентификатор в `HTML` придумали свойство `id`, которое можно назначить любому элементу `DOM`. Добавим свойство `id` к тегу `<p>`:

```
1  ...
2  <p id="text">Текст</p>
3  ...
```

Мы пометили наш элемент `идентификатором`. **Запомни – это очень важно: в одном документе не может быть несколько элементов с одинаковым id**. Иначе это приведет к ошибкам и непредсказуемым последствиям при выполнении кода.

Теперь, когда мы его отметили, нам нужно его выбрать с помощью **JavaScript**.

Ты должен был заметить, что внутри `HTML`-документа, я подключил файл `js`:

```
<script src="main.js"></script>
```

Открываем файл и пишем следующий код:

```
let text = document.getElementById('text');
```

Этот код означает, что в **HTML**-документе мы ищем элемент с `id` равным значению `text`. Если разжевать более подробно, то слушай сюда:

`document.` – означает, что мы ищем элемент в документе.

`getElementById` – читай, как "**получить элемент с id...**" и в скобках указываешь с каким `id`. Все предельно просто.

Метод `getElementById` всегда вернет тебе нужный элемент, если ты все правильно сделал. Если же ты где-то накосячил, то этот метод вернет тебе значение `null`.

Переходим ко 2-у вопросу: `добавление зеленого цвета`. Дополняем код в файле `main.js`:

```
1 let text = document.getElementById('text');
2 text.style.color = 'green';
```

Вот и все. Одна строка, и цвет текста поменялся. Записав в переменную `text` найденный элемент мы можем делать с ним что угодно.

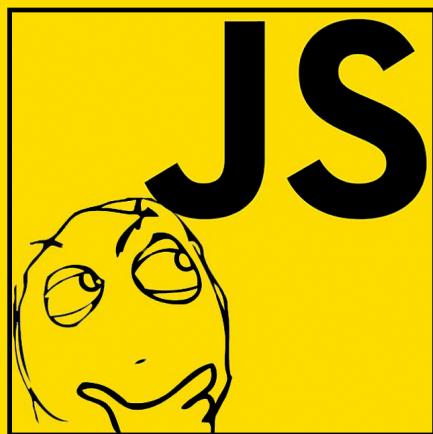
В данном же случае, мы фактически поменяли стиль текста, т.к. цвет относится именно к стилевому оформлению. Поэтому так и написали: `text.style.color` и указали цвет, который нужно применить.

О всех возможных действиях над элементом буду повествовать тебе по мере обучения.

DOM-элементы. События. Ввод.

JavaScript для тупых

DOM-элементы События. Ввод



@javascriptstupid

Ты даже не представляешь, как много всего происходит в браузере, когда ты производишь какие-то действия: кликаешь ли мышкой, причём браузер различает, правой и/или левой мышью ты кликнул, отслеживает поведение колесика твоей мышки (представляешь?), отслеживает над каким элементом ты сейчас водишь мышкой и т.д и т.п.

Поэтому в этом уроке я хочу начать разбирать тему событий .

Создание странички

Давай разберем событие клика по кнопке. Создаем html-страницу, на которой будет текст и 2 кнопки:

Я текст, над которым издеваются

Покрасить текст в красный

Покрасить текст в зеленый

Код этой страницы:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width">
6          <title>repl.it</title>
7          <link href="style.css" rel="stylesheet" type="text/css"/>
8      </head>
9      <body>
10         <p id="text">Я текст, над которым издеваются</p>
11
12         <button id="btnRed">Покрасить текст в красный</button>
13
14         <button id="btnGreen">Покрасить текст в зеленый</button>
15
16         <script src="script.js"></script>
17     </body>
18 </html>
```

Итак, что мы здесь имеем? Мы имеем тэг `p` с `id="text"` и 2 кнопки: одна с `id="btnRed"`, а другая с `id="btnGreen"`. При нажатии на кнопку, текст в тэге `p` должен перекрашиваться в соответствующий цвет, который указан на кнопке.

Выбор нужных элементов

Что нам необходимо? Нам необходимо с помощью **JavaScript** получить 3 элемента для управления и обработки событий: тэг `p`, и 2 тэга `button`. Пишем в файле `script.js` свой **JS**-код:

```
1 let text = document.getElementById('text');
```

```
2 let btnRed = document.getElementById('btnRed');  
3 let btnGreen = document.getElementById('btnGreen');
```

Так что мы сделали? Правильно, мы с помощью **JavaScript** нашли требуемые нам элементы и теперь можем с ними работать. А что произойдет, если ты не разместил элемент на странице или ошибся с указанием `id`? В значение переменной **JavaScript** запишет значение `null`. Поэтому будь внимательным.

Детектируем событие click

Что нам необходимо сделать дальше? Дальше нам необходимо как-то объяснить браузеру, что мы хотим выполнить некоторые действия при нажатии на кнопки. Два самых популярных способа сделать это средствами **JavaScript**: использовать метод `onclick` у элемента `button`, либо использовать метод `addEventListener`.

Метод onclick

К коду:

```
1 btnRed.onclick = function() {  
2     text.style.color = 'red';  
3     alert('Цвет текста изменен на красный');  
4 };
```

Сначала о том, что здесь происходит. Как только произойдет клик по кнопке с `id="btnRed"`, выполнится код, находящийся внутри указанной функции, т.е.:

```
1 text.style.color = 'red';  
2 alert('Цвет текста изменен на красный');
```

У элемента с `id="text"` поменяется цвет текста на красный и появится сообщение, которое сообщит об этом пользователю.

Метод `onclick` вызывается непосредственно у того элемента, событие которого мы хотим обрабатывать. В данном случае, мы обрабатываем событие клика у элемента `btnRed`. Синтаксис достаточно простой. Все, что нужно – это определить метод `onclick`, т.е. задать функцию, которая будет выполняться в том случае, если на элемент, в данном случае, кнопку `btnRed` кликнут мышкой (замечу, что `onclick` отработает только при клике по элементу **левой кнопкой мыши**).

В данном случае, мы указали стандартную функцию, но можно указать и стрелочную:

```
1 btnRed.onclick = () => {
2   text.style.color = 'red';
3   alert('Цвет текста изменен на красный');
4 };
```

Все будет работать точно так же. Какой вариант лучше? Т.к. я больше пользуюсь возможностями `ES6`, то и пишу я в таких случаях чаще всего стрелочную функцию, т.е. мой выбор – 2 вариант. Аналогично этому обработчику события ты должен создать еще один такой же обработчик, но для другой кнопки. По коду изменится только цвет, который ты хочешь выставить. Итого, файл `script.js` должен иметь примерно следующий вид:

```
1 let text = document.getElementById('text');
2 let btnRed = document.getElementById('btnRed');
3 let btnGreen = document.getElementById('btnGreen');
4
5 btnGreen.onclick = () => {
6   text.style.color = 'green';
7   alert('Цвет текста изменен на зеленый');
8 };
9
10 btnRed.addEventListener('click', () => {
11   text.style.color = 'red';
12   alert('Цвет текста изменен на красный');
13 });
```

Я намерено использовал разные методы для обработки события `click`, чтобы показать оба.

Метод `addEventListener`

Кроме метода `onclick` очень часто для обработки тех же кликов можно увидеть следующую конструкцию, которая работает ровно так же:

```
1 btnRed.addEventListener('click', () => {
2   text.style.color = 'red';
3   alert('Цвет текста изменен на красный');
4});
```

Синтаксис также очень похож на предыдущий метод, за исключением того, что в `onclick` мы записываем конкретную функцию, которая вызовется при отработке события `click`, а здесь мы передаем внутрь метода 2 аргумента `addEventListener : название события`, которое мы хотим слушать у элемента и **функцию**, которая выполнится, когда данное событие произойдет. Здесь также можно использовать стандартную функцию, описание которой начинается с ключевого слова `function`, либо, снова используем стрелочную функцию, как я и сделал в данном примере.

Разница между `onclick` и `addEventListener`

Разница на самом деле в том, что `addEventListener` более универсальный метод, который может принимать в качестве аргумента огромный список событий (посмотреть все события можно [здесь](#)). Т.е., этот метод общий для нескольких десятков событий.

Метод `onclick` – обрабатывает только клик по кнопке и больше ничего.

Хочу чтобы ты обратил внимание на то, что сокращенная запись событий есть не только у клика (`onclick`), но и практически у всех других событий, которые может отдавать элемент.

Улучшим наш код

Просто ради того, чтобы зацепить побольше пройденных тем, я хочу улучшить тот код, который у нас получился ранее.

Первое улучшение

Во-первых, обрати внимание на то, как ты получаешь элементы.

```
1 let text = document.getElementById('text');
2 let btnRed = document.getElementById('btnRed');
3 let btnGreen = document.getElementById('btnGreen');
```

Чем отличаются эти 3 строки? Правильно, только **названием переменных и id**, которые мы передаем в метод `getElementById`. Не кажется ли тебе, что эти строки слишком длинные для того чтобы их повторять? Давай улучшим это. Для этого добавим свою функцию `getElement`:

```
1 function getElement(idElement) {
2     return document.getElementById(idElement);
3 }
```

Надеюсь, ты понимаешь, что делает эта функция. **Если нет, я повторю:** функция принимает как аргумент `id` элемента, который ты хочешь найти в документе и возвращает тебе значение, которое возвращает метод `getElementById`. Теперь наш переработанный код будет выглядеть так:

```
1 let text = getElement('text');
2 let btnRed = getElement('btnRed');
3 let btnGreen = getElement('btnGreen');
```

Как видишь, теперь запись стала короче, но, при этом смысл абсолютно не поменялся. Более того, если вдруг, ты решишь поменять поведение процесса выбора элемента, то тебе не нужно будет исправлять это во всем коде, а лишь

поправить только в функции `getElement` и во всех остальных частях кода это автоматически применится. Круто, не правда ли?

Второе улучшение

Внутрь нашей функции `getElement` мы передаем `id`-элемента, который мы хотим получить. И передаем мы его явной строкой. Но давай подумаем, а удобно ли это? Представь, если у нас в коде будет не как сейчас 20 строк, а хотя бы 300-400, удобно ли будет искать название `id` во всем коде? Наверное нет. Поэтому у программистов есть **негласное правило** – *все строковые значения, которые никак не будут меняться по ходу написания программного кода нужно выносить в отдельные переменные, так называемые константы.*

Константы представляют собой обычные переменные, которые инициализируются с помощью ключевого слова `const` (неудивительно, да?). Но есть одно отличие от обычных переменных в именовании константы. Ты должен назвать ее с помощью включенного `CAPS LOCK` =) Т.е., переменная должна иметь следующий вид:

```
const TEXT_ID = 'text';
```

И в разделении слов используется стиль не `camelCase` (когда одно слово заканчивается, а второе начинается с заглавной буквы), а стилем `snake case` (когда одно слово в названии переменной разделяется от другого нижним подчеркиванием).

Итак, переведем все наши `строковые значения` в `константы`. Кроме `id` элементов у нас это еще и цвета, не забываем и про них. Итого получаем:

```
1 const TEXT_ID = 'text';
2 const BTN_RED = 'btnRed';
3 const BTN_GREEN = 'btnGreen';
4 const RED_COLOR = 'red';
5 const GREEN_COLOR = 'green';
6 const TEXT_CHANGE_TO_GREEN = 'Цвет текста изменен на зеленый';
7 const TEXT_CHANGE_TO_RED = 'Цвет текста изменен на красный';
8
```

```
9  function getElement(idElement) {  
10    return document.getElementById(idElement);  
11  }  
12  
13  let text = getElement(TEXT_ID);  
14  let btnRed = getElement(BTN_RED);  
15  let btnGreen = getElement(BTN_GREEN);  
16  
17  btnRed.onclick = function() {  
18    text.style.color = RED_COLOR; // тоже заменяем во всех\  
19    // методах на нужную константу  
20    alert(TEXT_CHANGE_TO_RED);  
21  };  
22  
23  btnGreen.addEventListener('click', () => {  
24    text.style.color = GREEN_COLOR;  
25    alert(TEXT_CHANGE_TO_GREEN);  
26  });
```

В реальных проектах константы определяются в отдельных файлах, либо в самом верху всего листинга кода. В результате этого, в случае изменения названия `id`, ты очень быстро найдешь нужную константу и изменишь ее значение. Снова удобство и снова плюс к скорости разработки.

Третье улучшение

Пойдем еще дальше. Наверное ты заметил, что внутри каждого метода мы устанавливаем цвет текста такой строкой:

```
text.style.color = GREEN_COLOR;
```

Знаешь на что я намекаю? На то, что нужно создать еще одну функцию, которая будет принимать значение цвета и устанавливать его для элемента с `id="text"`.

```
1  function setTextColor(color) {  
2    text.style.color = color;  
3  }
```

Снова меняем наш код и получаем:

```
1 const TEXT_ID = 'text';
2 const BTN_RED = 'btnRed';
3 const BTN_GREEN = 'btnGreen';
4 const RED_COLOR = 'red';
5 const GREEN_COLOR = 'green';
6 const TEXT_CHANGE_TO_GREEN = 'Цвет текста изменен на зеленый';
7 const TEXT_CHANGE_TO_RED = 'Цвет текста изменен на красный';
8
9 function getElement(idElement) {
10     return document.getElementById(idElement);
11 }
12
13 function setTextColor(color) {
14     text.style.color = color;
15 }
16
17 let text = getElement(TEXT_ID);
18 let btnRed = getElement(BTN_RED);
19 let btnGreen = getElement(BTN_GREEN);
20
21 btnRed.onclick = function() {
22     setTextColor(RED_COLOR); // тоже заменяем во всех \
23     // методах на нужную константу
24     alert(TEXT_CHANGE_TO_RED);
25 };
26
27 btnGreen.addEventListener('click', () => {
28     setTextColor(GREEN_COLOR);
29     alert(TEXT_CHANGE_TO_GREEN);
30});
```

Теперь, когда мы вынесли изменение цвета в отдельную функцию, в случае изменения логики, изменения цвета мы также поменяем эту логику только внутри самой функции.

Домашнее задание

Даю ссылку на [код](#).

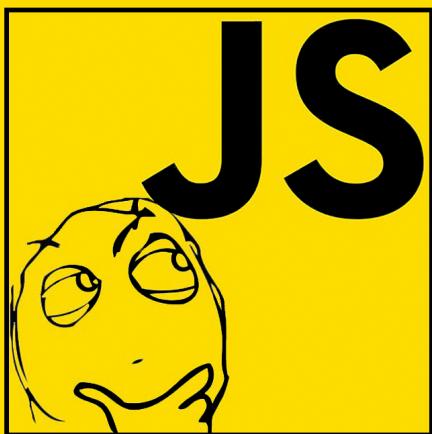
Твоя задача:

- полностью разобраться с тем, что там происходит;
- добавить свою кнопку и обработать событие клика, которая будет **перекрашивать текст в желтый**.

DOM-элементы. События. Часть 2

JavaScript для тупых

DOM-элементы События. Часть 2



@javascriptstupid

В предыдущем уроке мы разобрались, как можно **обрабатывать событие клика**, используя методы элемента `onclick` и `addEventListener`.

В этом уроке я хотел бы подсказать тебе, как посмотреть **все события**, которые ты можешь **отслеживать у элемента**.

Для просмотра событий элемента сначала создадим себе тестовую площадку.

Создай на своем компьютере папку, например, с названием `web`. В этой папке создай 3 файла: `index.html` , `script.js` , `style.css`

Содержимое файла `index.html` :

```
1  <!DOCTYPE html>
2  <html>
3    <head>
```

```
4      <meta charset="utf-8">
5      <meta name="viewport" content="width=device-width">
6      <title>Page</title>
7      <link href="style.css" rel="stylesheet" type="text/css" />
8  </head>
9  <body>
10     <button class="red-btn" id="btnRed">Большая красная \
11         кнопка</button>
12
13     <script src="script.js"></script>
14 </body>
15 </html>
```

Содержимое файла `style.css` :

```
1 .red-btn {
2     background: red;
3     height: 100px;
4     width: 250px;
5     font-size: 16px;
6     color: white;
7 }
```

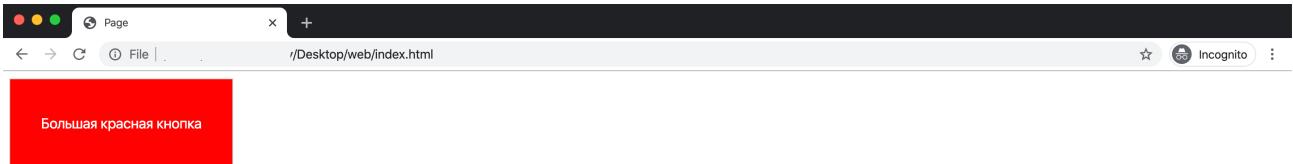
Это файл стилей. Все что мы делаем здесь: меняем цвет кнопки на красный, высоту кнопки (`height`) на 100 пикселей, ширину (`width`) на 250 пикселей, устанавливаем размер шрифта (`font-size`) в 16px, и цвет текста (`color`) в кнопке на белый. Если хочешь больше узнать о CSS, то придется гуглить. Здесь мы изучаем **JavaScript** =)

Содержимое файла `script.js` :

```
1 let btnRed = document.getElementById('btnRed');
2
3 console.dir(btnRed);
4
5 btnRed.onclick = function() {
6     alert('На меня нажали левой кнопкой мыши');
7 };
8
9 btnRed.oncontextmenu = function() {
10     alert('На меня нажали правой кнопкой мыши');
```

```
11 };
```

Когда ты создаешь все файлы с указанным содержимым открай с помощью своего браузера файл `index.html`. Должно получится следующее:



Как ты понимаешь, нас больше всего интересует файл с **JS-кодом**. Давай его разберем.

Первым делом, как обычно, ты должен выбрать элемент с которым хочешь взаимодействовать с помощью **JavaScript**, в данном случае это кнопка с `id="btnRed"`:

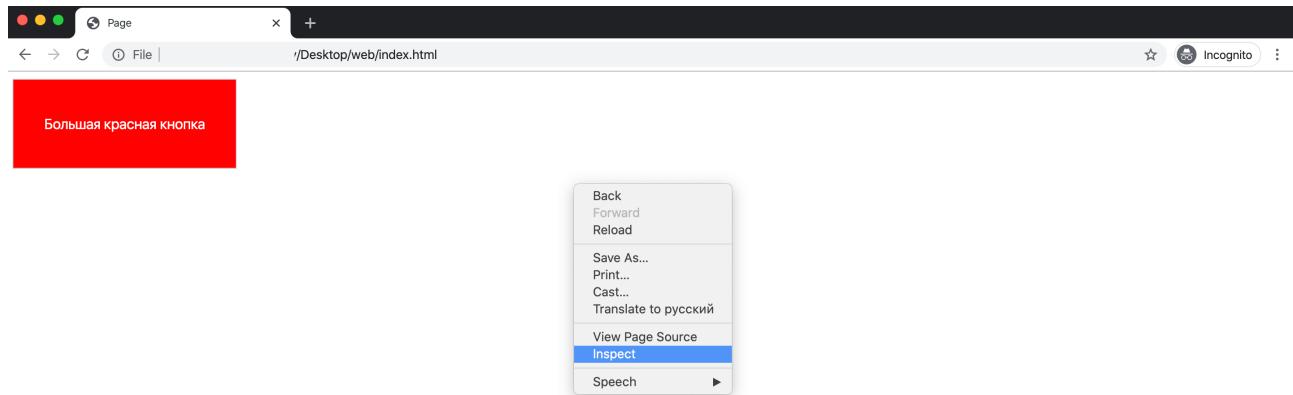
```
let btnRed = document.getElementById('btnRed');
```

Дальше идет строка:

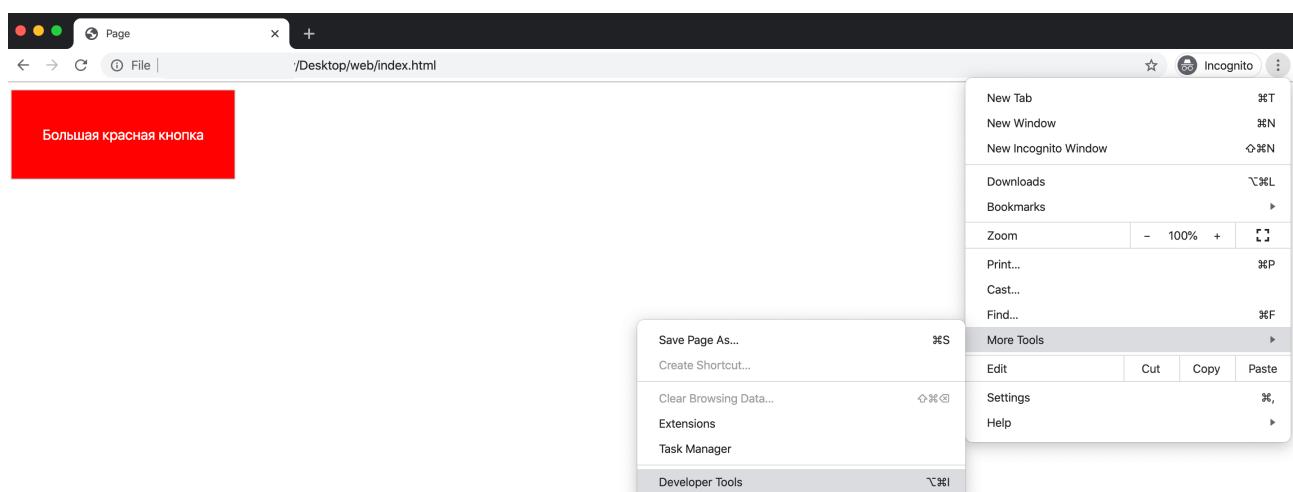
```
console.dir(btnRed);
```

Ты привык видеть `console.log()`, который выводит текстовое сообщение в консоль, а `console.dir()`, в свою очередь, выводит список свойств указанного JS-объекта (наша кнопка является DOM-объектом).

Итак, чтобы посмотреть, что же выводит в консоль данная команда нам нужно открыть **инструменты разработчика (Developer Tools)**. А для этого делаем следующее:



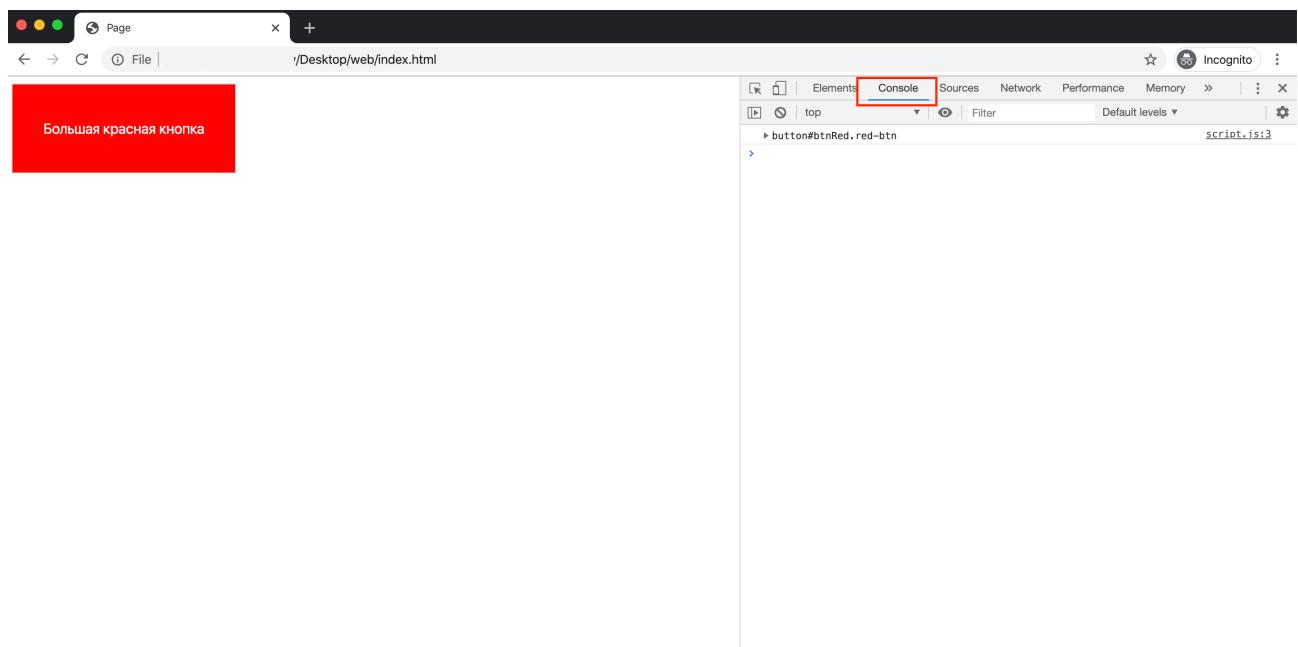
Либо так:



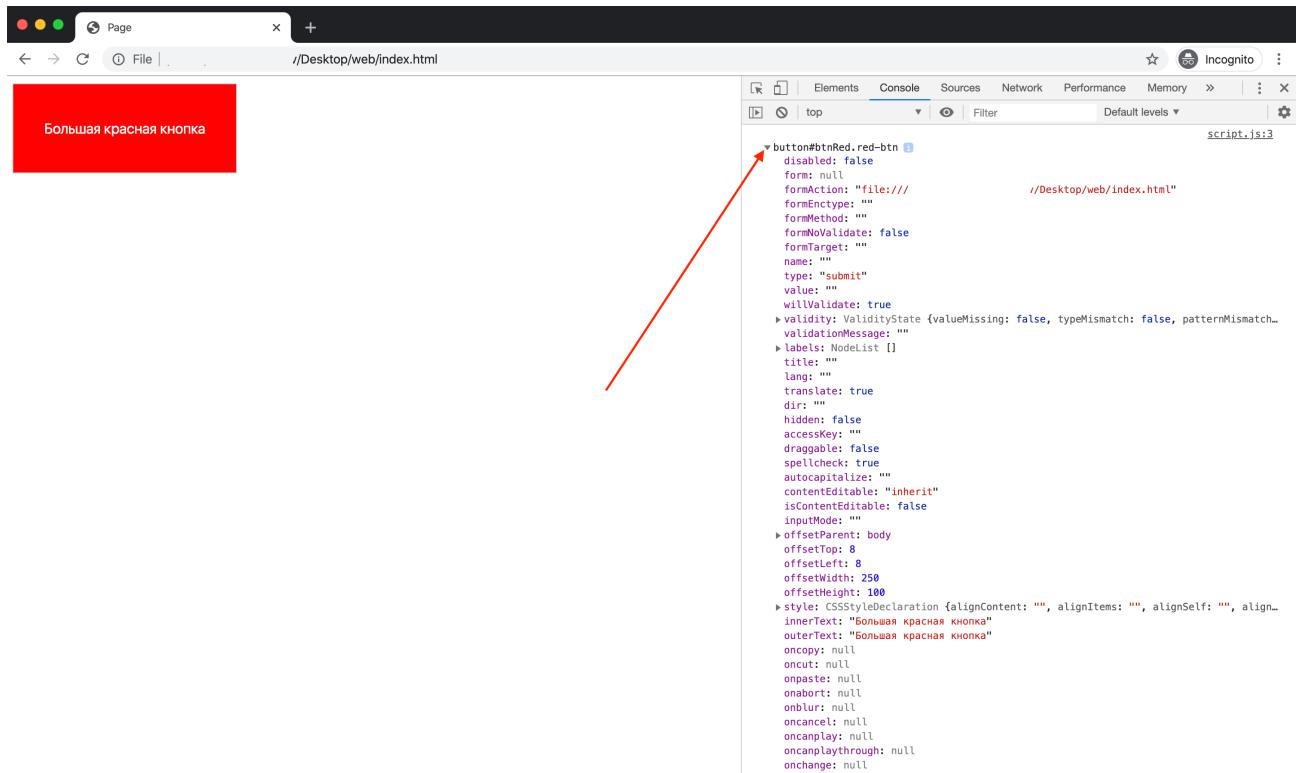
Либо, горячими клавишами. Но они отличаются от браузера к браузеру, от операционной системы к операционной системе.

Как ты заметил у меня все англоязычное и поэтому интуитивно ищи данные менюшки в своем браузере, если он у тебя на русском или любом другом языке. Если не получается найти, то гугли)

После открытия данной панели нам нужно выбрать консоль :



Если ты все сделал правильно, то у тебя там уже будет выведена надпись, но она не простая, а иерархическая. Это означает, что мы можем ее раскрыть нажав на стрелочку слева от нее:



И здесь открывается огромный список различных свойств и методов. Но мы ищем только те методы, которые начинаются со слова `on`. Дело в том, что методы, которые начинаются с `on`, являются событиями, можно обрабатывать у данного элемента. Список примерно следующий:

```
oncopy: null
oncut: null
onpaste: null
onabort: null
onblur: null
oncancel: null
oncanplay: null
oncanplaythrough: null
onchange: null
▶ onclick: f ()
onclose: null
▶ oncontextmenu: f ()
oncuechange: null
ondblclick: null
ondrag: null
ondragend: null
ondragenter: null
ondragleave: null
ondragover: null
ondragstart: null
ondrop: null
ondurationchange: null
onemptied: null
onended: null
onerror: null
onfocus: null
oninput: null
oninvalid: null
onkeydown: null
onkeypress: null
onkeyup: null
onload: null
onloadeddata: null
onloadedmetadata: null
onloadstart: null
onmousedown: null
onmouseenter: null
onmouseleave: null
onmousemove: null
onmouseout: null
onmouseover: null
onmouseup: null
onmousewheel: null
onpause: null
onplay: null
onplaying: null
```

```
onprogress: null
onratechange: null
onreset: null
onresize: null
onscroll: null
onseeked: null
onseeking: null
onselect: null
onstalled: null
onsubmit: null
onsuspend: null
ontimeupdate: null
ontoggle: null
onvolumechange: null
onwaiting: null
onwheel: null
onauxclick: null
ongotpointercapture: null
onlostpointercapture: null
onpointerdown: null
onpointermove: null
onpointerup: null
onpointercancel: null
onpointerover: null
onpointerout: null
onpointerenter: null
onpointerleave: null
onselectstart: null
onselectionchange: null
onanimationend: null
onanimationiteration: null
onanimationstart: null
ontransitionend: null
```

Как видишь, их огромное множество и разбирать их все со мной – это гиблое дело. Общий прогресс не будет идти. Поэтому, посмотри сам, что они делают, как и зачем. Полный список событий с описанием находится [здесь](#).

Для примера я обработал 1 событие (нажатие правой кнопкой по элементу) разными способами:

```
1 btnRed.oncontextmenu = function() {
2     alert('На меня нажали правой кнопкой мыши');
3 };
4
5 // либо так
6 btnRed.addEventListener('contextmenu', function() {
7     alert('На меня нажали правой кнопкой мыши');
```

```
8  });
```

Как видишь, в первом случае вызывается метод `oncontextmenu`, а во втором в метод `addEventListener` передается название события, но без начальной приставки `on` – `contextmenu`. Это правило касается **всех событий**.

Остальные события пробуй изучать сам. Потому что все обработчики для них пишутся аналогично.

Домашнее задание

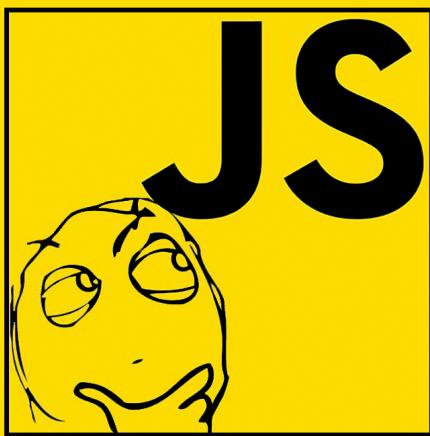
Почитай о других событиях и попробуй сделать для них обработчик в соответствии с примером выше.

[Читать о других событиях здесь](#)

Объекты. Начало игры

JavaScript для тупых

Объекты Начало игры ^-^



@javascriptstupid

Вначале **объекты** могут показаться непонятной и ненужной фигней, но, поверь, это далеко не так. Объекты очень крутые ^_^ и будут выручать тебя не раз. Поэтому, эту тему нужно понять максимально хорошо.

Объекты выглядят следующим образом:

```
1 const boy = {  
2     name: 'Вася',  
3     age: 12  
4 };
```

Объект всегда имеет такой вид. После знака = мы должны открыть фигурные скобки {} и весь код писать внутри них.

`name` и `age` называются **свойствами объекта**. В свойства можно записывать данные любых типов: строки, числа, булевые значения, другие объекты, массивы (о них в следующих уроках).

Но объекты были бы не настолько полезны, если бы не было возможности добавить в них методы. **Методы** – это функции, которые могут выполнять данный объект. Сейчас все объясню, ты удивишься насколько просто это запомнить =)

Сначала, добавлю метод в существующий объект:

```
1 const boy = {  
2     name: 'Вася',  
3     age: 12,  
4  
5     sayHello: function() {  
6         alert('Привет, меня зовут', this.name);  
7     }  
8 };
```

Хочу заметить, что после каждого свойства нужно ставить обязательно запятую, как и после каждого метода, иначе будет ошибка.

Что мы можем теперь сделать с этим объектом? Мы можем запросить любое его свойство или метод:

```
1 const boy = {  
2     name: 'Вася',  
3     age: 12,  
4  
5     sayHello: function() {  
6         alert('Привет, меня зовут', this.name);  
7     }  
8 };  
9  
10 alert('Мое имя', boy.name);  
11 alert('Мой возраст', boy.age);  
12 boy.sayHello(); // выведет alert, который внутри этого \  
13 // метода  
14  
15 alert('Мое имя', boy['name']);
```

```
16 alert('Мой возраст', boy['age']);  
17 boy['sayHello'](); // выведет alert, который внутри этого \  
18 // метода
```

К **свойству** или **методу** можно обращаться через **точку**, а можно как к элементам массива (о них в следующем уроке). Но, на практике, все-таки, чаще применяют подход вызова через точку. **Но у этого вызова есть важный нюанс – свойства и методы должны быть одним словом (без пробелов)**. Обычно методы имеют название в 2 или 3 слова (как и в нашем случае), поэтому их называют в стиле `camelCase`, в нашем случае `sayHello`, а не `say hello`, или `say_hello`.

Итак, чем же отличаются **свойства от методов** и зачем, вообще, это все нужно?

JavaScript – объектно-ориентированный язык. Может ты где-то читал об этом раньше или слышал. Так вот, объектно-ориентированного подхода в программировании придерживаются почти все программисты. Практически все популярные ЯП являются **объектно-ориентированными**.

*Если говорить простыми словами, то это подход при котором **всё с чем ты работаешь нужно представлять в виде объектов**.*

А мы живем с тобой в мире **объектов**: компьютер или телефон, с которого ты читаешь данный урок является **объектом**. У него есть **свойства**:
`марка, модель, разрешение экрана;` у него есть **методы (функции)**:
`звонить, писать, отправлять/принимать какие-то данные и т.д. и т.п.`

Надеюсь ты начинаешь понимать что к чему. В каком-то смысле, данный подход облегчает разработку.

Немного соберу все вместе. *Внимательно прочитай и запомни это, чтобы никогда не путать свойства с методами и не путать их предназначение.*

Немного практики

Давай мы с тобой замутим объект, который будет представлять собой героя в компьютерной игре.

Итак, что мы знаем о героях из компьютерных игр? У них должны быть минимальные **характеристики**: здоровье, атака, защита, накопленный опыт, уровень и для того, чтобы он умел ходить, добавим ему в **свойства** значение координаты, в которой он находится. Ну, и, конечно же, придумаем ему имя. Приступим:

```
1 const hero = {  
2     name: 'jsHero',  
3     attack: 15,  
4     defense: 5,  
5     level: 1,  
6     experience: 0,  
7     coordinate: 0  
8 };
```

Мы получили героя с именем `jsHero` и прописали ему характеристики, которые в **объекте** являются **свойствами**. Героя мы получили, но вот ни ходить, ни драться, не защищаться, да и, вообще, ничего он делать сейчас не умеет. Он просто стоит и всё. Давайте научим его хотя бы ходить.

Не будем усложнять и сделаем так, чтобы наш герой умел ходить только налево и направо. Вспомните игрушку из детства `Super Mario`:



Будем считать, что значение `0`, в свойстве `coordinate` соответствует самому левому положению героя. Т.е, если герою нужно будет идти `вправо`, то координата будет каждый **шаг увеличиваться на единицу**, а `влево` – **уменьшаться на единицу**. Итак, создаем методы, которые научат нашего героя ходить:

```
1 const hero = {  
2     name: 'jsHero',  
3     attack: 15,  
4     defense: 5,  
5     level: 1,  
6     experience: 0,  
7     coordinate: 0,  
8  
9     goRight: function() {  
10         this.coordinate++;  
11     },  
12  
13     goLeft: function() {  
14         this.coordinate--;  
15     }  
16 };
```

Теперь наш герой умеет ходить. Для этого нам просто нужно вызвать один из 2-х методов:

```
1 hero.goRight(); // для того, чтобы идти вправо  
2 // (вперед в нашей системе координат)  
3 hero.goLeft(); // для того, чтобы идти влево  
4 // (назад в нашей системе координат)
```

Внутри этих методов наверное ты заметил обращение к какой-то непонятной переменной `this.coordinate`. С помощью `this` мы можем обращаться к свойствам и методам объекта внутри самого объекта. Разберём:

```
1 ...  
2 goRight: function() {  
3     this.coordinate++;
```

```
4  }
5  ...
```

Чтобы внутри метода `goRight` обратиться к свойству `coordinate` этого же объекта нужно обязательно использовать ключевое слово `this`. Если не сделать этого, то получится неприятный сюрприз в виде ошибки:

```
1  ...
2  goRight: function() {
3      coordinate++; // ReferenceError: coordinate is not \
4      // defined
5  }
6  ...
```

Ошибка гласит о том, что переменная `coordinate` **не определена**, что и логично. Ведь нигде у нас в коде не существует данной переменной. Существует только одноимённое свойство в объекте и его нужно вызывать только через `this.coordinate`.

Итак, ходить научили, давайте для закрепления научим героя **атаковать** и **защищаться**:

```
1  const hero = {
2      name: 'jsHero',
3      attack: 15,
4      defense: 5,
5      level: 1,
6      experience: 0,
7      coordinate: 0,
8
9      goRight: function() {
10          this.coordinate++;
11      },
12
13      goLeft: function() {
14          this.coordinate--;
15      },
16
17      hit: function() {
```

```
18     this.experience += 10;
19     alert('Ударил врага');
20 },
21
22     blockKick: function() {
23     alert('Заблокировал удар');
24 }
25 };
```

Теперь для атаки используем этот метод:

```
hero.hit(); // атака
```

За каждый удар, мы будем начислять очки опыта увеличивая значения свойства `experience` на `10` единиц.

А для защиты этот:

```
hero.blockKick(); // защита
```

Ссылка на живой пример ([тык](#)). Попробуй дописать свои методы и свойства. Вызови их и поделись результатами с другими в комментариях.

Итак, подведу небольшой итог.

Свойства

В свойства записываются какие-либо **данные**: текст, числа, булевые значения, даты, массивы, другие объекты.

Методы

Методы предназначены для того, чтобы что-то сделать со **свойствами**: как-либо изменить, дополнить, удалить и так далее.

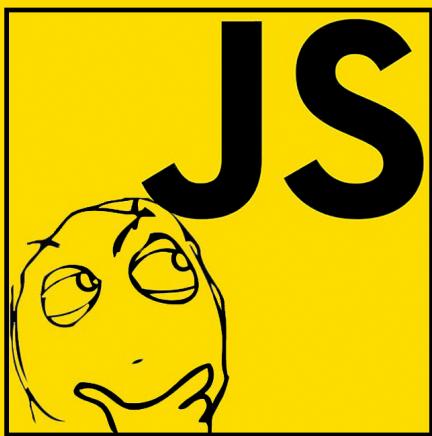
Ключевое слово this

Используется для обращения к **свойствам и методам** внутри самого **объекта**.

Массивы

JavaScript для тупых

Районы, кварталы, жилые массивы...



@javascriptstupid

Массивы требуются очень часто, настолько часто, что это, пожалуй, второй по частоте использования **типа данных**, которыми оперируют в профессиональной разработке.

Чтобы было проще понять что это и зачем это нужно я приведу пример. Представь, что у нашего героя будет несколько видов оружия, а возможно, мы сделаем так, что можно будет подбирать новое оружие. К примеру, у нашего героя имеются: меч, секира, булава, лук. Где хранить эту информацию?

Именно для таких случаев и придумали **массивы**. Массив описывается следующим образом:

```
1 const arr = new Array();  
2 const arr2 = [];
```

Существует **2 способа** записи. Но я рекомендую тебе использовать всегда второй. И это не то чтобы лично моя рекомендация, а **лучшая практика среди разработчиков**.

Массивы – это списки. При этом списки из чего угодно. Это может быть список из строковых значений :

```
const arr = ['ручка', 'карандаш'];
```

цифровых значений :

```
const arr = [1, 2, 3, 4, 5];
```

из объектов :

```
1 const arr = [
2     {name: 'Name 1', value: 1},
3     {name: 'Name 2', value: 2}
4 ];
```

Словом, из всего того, что ты можешь придумать.

Массивы очень удобны по нескольким причинам: в них легко **добавлять/удалять/изменять/искать/перебирать элементы**.

Длина массива

У каждого массива существует свойство `length`. Если запросить это свойство, то оно вернёт количество элементов содержащихся в массиве:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];
2
```

```
3 arr.length; // 3
```

Получение элемента из массива

Каждый элемент массива имеет свой идентификатор (**индекс**). Это означает, что, фактически каждый элемент в массиве нумеруется. Нумерация начинается не с единицы, как ты привык в обычной жизни, а с **нуля**. Чтобы выбрать нужный элемент из массива, нужно указать его **индекс**:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];
2
3 arr[0]; // ручка
4 arr[1]; // карандаш
5 arr[2]; // ластик
6 arr[3]; // undefined
```

Изменение элемента в массиве

Изменить элемент массива очень просто. Выбираешь элемент по **индексу** и присваиваешь нужное **значение**. Вот собственно и всё:

```
1 let arr = ['ручка', 'карандаш', 'ластик'];
2
3 arr[0] = 'пластелин';
4
5 console.log(arr); // ['пластелин', 'карандаш', 'ластик']
```

Добавление элемента в массив

В любое время можно добавить новый элемент в массив и для этого существует достаточно много способов.

```
1 const arr = ['вилка'];
2
```

```
3 arr[1] = 'ложка';
4
5 arr; // ['вилка', 'ложка']
```

Как видишь, я просто обратился к несуществующему индексу в массиве и присвоил в него значение, тем самым добавив элемент в этот самый массив. Таким способом **НЕ рекомендуется** добавлять элементы, т.к. это может привести к ситуации, когда по ошибке можно написать примерно так:

```
1 const arr = [];
2 arr[0] = 'трололо';
3 arr[1] = 'лол';
4 arr[24] = 'kek';
```

Ты просто можешь по ошибке (или специально) вписать индекс больше количества элементов в массиве. Тут кроется свинка, так как если после кода выше запросить длину массива, то ты ужаснешься, ибо вместо ожидаемого значения 3, ты увидишь значение 25:

```
1 const arr = [];
2 arr[0] = 'трололо';
3 arr[1] = 'лол';
4 arr[24] = 'kek';
5
6 arr.length; // 25
```

Дело в том, что `length` на самом деле не считает количество элементов, а просто выдает наибольший цифровой индекс в массиве и прибавляет к нему единицу.

Следующий способ добавления можно назвать **классическим**:

```
1 const arr = [];
2
3 arr.push('вилка');
```

```
4 arr.push('ложка');
5
6 arr; // ['вилка', 'ложка']
```

Метод `push` добавляет элементы в **конец массива**. Это является классической ситуацией, когда мы добавляем элемент в **конец массива**, но бывают ситуации, когда требуется вставить элемент в **начало массива**. Для данной операции существует метод `unshift`:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];
2
3 arr.unshift('циркуль');
4
5 arr; // ['циркуль', 'ручка', 'карандаш', 'ластик']
```

Удаление элементов

Для удаления элементов существует тоже не один способ.

Для того, чтобы удалить элемент **из конца массива**, нужно воспользоваться методом `pop`:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];
2
3 arr.pop();
4
5 arr; // ['ручка', 'карандаш']
```

Интересным моментом является то, что метод `pop` возвращает удаленный элемент, т.е. если ты хочешь узнать какой элемент был удален, то просто запиши результат метода `pop` в переменную:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];
2
```

```
3 const delElement = arr.pop(); // 'ластик'  
4  
5 arr; // ['ручка', 'карандаш']
```

Для удаления элемента из начала массива существует метод `shift`, который подобно `pop` возвращает удаленный элемент:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];  
2  
3 const delElement = shift(); // 'ластик'  
4  
5 arr; // ['ручка', 'карандаш']
```

Перебор элементов массива

Очень часто требуется перебирать элементы массива, например для того, чтобы вывести их пользователю.

Для этого можно использовать циклы `for` или встроенный метод `forEach`.

Цикл `for` с массивом (является старым способом):

```
1 const arr = ['ручка', 'карандаш', 'ластик'];  
2  
3 for (let i = 0; i < arr.length; i++) {  
4     console.log(arr[i]);  
5 }
```

Надеюсь код понятен. Мы запускаем цикл `for` с индекса `0` и бежим до значения `arr.length` (**не включая его**), т.е. до конца массива. И выводим значение каждого элемента, подставляя значение переменной `i` в качестве *индекса массива*.

Так же можно воспользоваться циклом `for ... of`:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];
2
3 for (let el in arr) {
4     console.log(el);
5 }
```

Единственный минус данного способа в том, что ты не сможешь получить индекс элемента в массиве. Однако, это требуется не всегда, и если нужно что-то сделать только с самим значением, то этот способ будет не плохим выбором.

Также можно перебрать массив с помощью встроенного метода массива `forEach`:

```
1 const arr = ['ручка', 'карандаш', 'ластик'];
2
3 arr.forEach((elem, index) => {
4     console.log(elem);
5 });
```

Этот метод требует передать функцию, которая будет выполняться для каждого элемента массива. В переданной функции, в качестве первого аргумента `elem` подставим текущий элемент массива, а в качестве второго `index` подставим индекс элемента.

Почему не стоит использовать `new Array()`

В самом начале я рекомендовал использовать `[]` для создания массива, но никак не `new Array()`.

Первая причина не использовать данный способ очень простой. Сравни и поймешь:

```
1 const arr = ['ручка', 'карандаш'];
2
3 const arr2 = new Array('ручка', 'карандаш');
```

Дело в том, что квадратные скобки намного короче в записи. А маленькое количества кода – почти всегда лучше, чем большое. А код начинает разрастаться с таких мелочей.

Вторая причина не использовать данный способ намного страшнее. Посмотри код:

```
1 const arr = new Array(50);
2
3 console.log(arr); // что выведет? (1)
4 console.log(arr.length); // что выведет? (2)
```

1. Ты наверное думаешь, что ответ `[50]` ?
2. Думаешь ответ `1` ?

А вот и нет, если `new Array()` принимает только один числовой аргумент, то он устанавливает это число в длину массива. Сам же не содержит ни одного элемента. Вызов `arr[0]` выдаст тебе значение `undefined`:

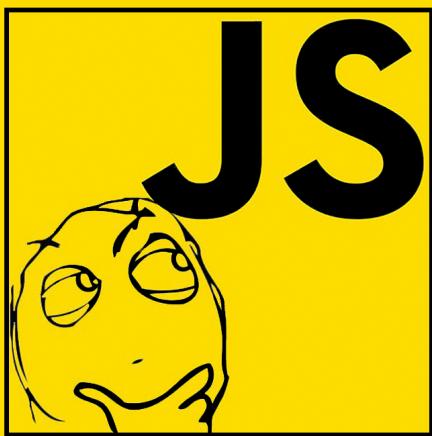
```
1 const arr = new Array(50);
2
3 console.log(arr); // [empty x 50]
4 console.log(arr[0]); // undefined
5 console.log(arr.length); // 50
```

Получается, что данный способ имеет *специфическое поведение*, а такие моменты программисты пытаются избегать.

Методы массивов: splice, slice, concat

JavaScript для тупых

Методы массивов splice, slice, concat



@javascriptstupid

В предыдущем уроке я рассказал тебе о массивах и базовых методах для работы с ними. В этом и последующих уроках мы разберем, практически, полный список методов, которые постоянно применяются в разработке.

Метод splice

В предыдущем уроке мы узнали как добавлять элементы в начало/конец массива , а так же как удалять элементы с конца/начала массива . Скорее всего ты задавался вопросом, а как же удалить элемент, если он не в начале массива и не в конце.

Для данного действия, существует метод `splice` , которым ты в будущем будешь пользоваться очень часто, т.к. этот метод умеет и добавлять и удалять элементы в любом месте массива. Переходим к примеру использования:

```
1 const languages = ['python', 'java', 'js'];
2 languages.splice(0, 2);
```

```
3 console.log(languages); // ['js']
```

Обрати внимание на аргументы при вызове `splice()`. Первый аргумент – `0`, а второй – `2`.

Первый аргумент – это указание индекса (позиции) элемента в массиве, с которого нужно начинать удаление.

Второй аргумент – количество элементов, которые нужно удалить с указанной позиции.

Итого, этим `languages.splice(0, 2)`, мы сказали: **удалили 2 элемента начиная с индекса 0**.

У этого метода есть еще одно достоинство – **умение добавлять элементы (любое количество) на место удаленных элементов**. Пример:

```
1 const languages = ['python', 'java', 'js'];
2 languages.splice(0, 2, 'php', 'c#', 'c++');
3 console.log(languages); // ['php', 'c#', 'c++', 'js']
```

Первые два аргумента метода объяснены выше, а вот следующие за ними аргументы будут вставляться в массив на место удаленных элементов. Их количество не ограничено – хочешь, добавь один элемент, хочешь 10 тысяч элементов, все будет прекрасно работать.

Этот метод меняет массив **по месту**, т.е. меняет исходный массив.

Метод slice

Метод копирует указанные элементы массива и создает из них новый.

```
1 const languages = ['java', 'python', 'php', 'c++', 'js'];
2 const newList = languages.slice(2, 4);
3
```

```
4 console.log(newList); // ['php', 'c++']
```

Метод `slice` не меняет исходный массив, а возвращает новый, поэтому для сохранения результатов его выполнения нужно эти самые результаты куда-то записать, для этого я определил константу `newList`.

Метод имеет только 2 аргумента.

Первый аргумент – индекс элемента С которого нужно начинать копирование.

Второй аргумент – индекс ДО которого нужно копировать элементы **не включая его**.

`languages.slice(2, 4)` – в данном случае, мы говорим: **скопириуй элементы с индексом 2 и 3. Элемент с индексом 4 не будет включен.**

Аргументы могут быть не только положительными числами, но и отрицательными, в таком случае отсчёт будет начинаться с конца массива.

Метод concat

Метод `concat`, фактически, создает из нескольких массивов/значений один, объединяя их. Результат выполнения метода – новый массив, который нужно куда-то записать для дальнейшей работы с ним.

```
1 const languages = ['python', 'java', 'js'];
2 const additionalLanguages = ['c#', 'c++'];
3
4 const summary = languages.concat(additionalLanguages);
5 console.log(summary); // ['python', 'java', 'js', 'c#', \
6 // 'c++']
```

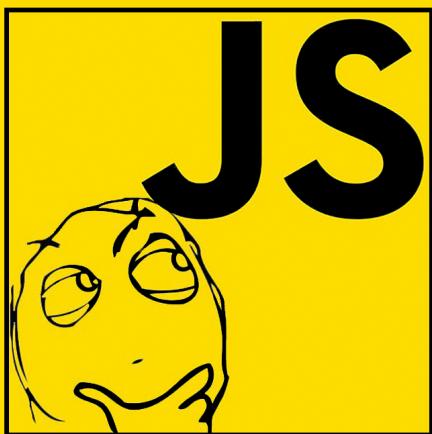
Метод принимает неограниченное количество аргументов. Если аргумент является массивом, то все его элементы копируются. В противном случае копируется сам аргумент.

```
1 const languages = ['js'];
2 languages.concat(['python', 'php'], 'java'); // ['js', \
3 // 'python', 'php', 'java']
```

Методы массивов: `find`, `findIndex`, `filter`

JavaScript для тупых

Методы массивов `find`, `findIndex`, `filter`



@javascriptstupid

Метод `find`

Метод `find` позволяет возвращать значение первого элемента, найденного в массиве, который удовлетворяет условию переданному в **callback-функцию**. Callback-функция запускается для каждого элемента в массиве. В случае, если элемент с указанным условием не найден, метод возвращает `undefined`.

Как только метод находит первый элемент удовлетворяющий условию в **callback-функции**, работа метода прекращается.

Этот метод посложнее тех, что рассмотрены ранее. Здесь в аргументах метода нужно передавать функцию. Но в этом ничего страшного или сложного нет. Метод `find` чаще всего используется с массивом, элементами которого являются объекты.

Пример:

```
1 const cars = [
2     {id: 1, model: 'Ferrari'},
3     {id: 2, model: 'Audi'},
4     {id: 3, model: 'Toyota'}
5 ];
6
7 const audi = cars.find(function(item, index, arr) {
8     return item.model === 'Audi';
9 });
10
11 console.log(audi); // { id: 2, model: 'Audi' }
```

В данном примере мы ищем объект в массиве `cars` у которого свойство `model` равно значению `Audi`.

В метод `find` мы передали целую функцию (`callback`). У этой функции 3 аргумента, 2 из последних используются редко.

Метод `find` последовательно перебирает все элементы массива и для каждого элемента запускает `callback-функцию`, которая должна вернуть `true/false`.

Первый аргумент – элемент массива, до которого в данный момент добрался метод `find`.

Второй аргумент – индекс этого самого элемента в массиве `cars`.

Третий аргумент – весь массив `cars` (т.е. массив по которому пробегает `find`).

Сейчас в реальных проектах чаще всего используются стрелочные функции в качестве аргумента метода `find`. Это удобнее и запись чаще всего оказывается намного короче. Тот же пример, но со **стрелочной функцией**. Так же, примите во внимание, что 2-й и 3-й аргументы метода в решении нашей задачи нам не нужны, следовательно, откинем их и получим следующее:

```
1 const cars = [
2     {id: 1, model: 'Ferrari'},
3     {id: 2, model: 'Audi'},
```

```
4     {id: 3, model: 'Toyota'}
5 ];
6
7 const audi = cars.find(item => item.model === 'Audi');
8
9 console.log(audi); // { id: 2, model: 'Audi' }
```

Запись стала намного короче и читается намного проще. В том случае, если метод `find` пробежал по всем элементам и не обнаружил ни одного элемента удовлетворяющего нашему условию (`item.model === 'Audi'`), то метод вернет значение `undefined`.

Метод `findIndex`

Метод `findIndex` работает так же, как и метод `find`, за исключением того, что возвращает не сам элемент, а его индекс.

Метод `filter`

Данный метод очень сильно похож на метод `find`, за тем исключением, что метод не завершает свою работу после первого найденного элемента, а продолжает ее, пока не пройдет по всему массиву. Из описания несложно догадаться, что метод возвращает массив элементов удовлетворяющих условию в **callback-функции**.

Давайте выберем все автомобили, у которых значение свойство `id` является четным:

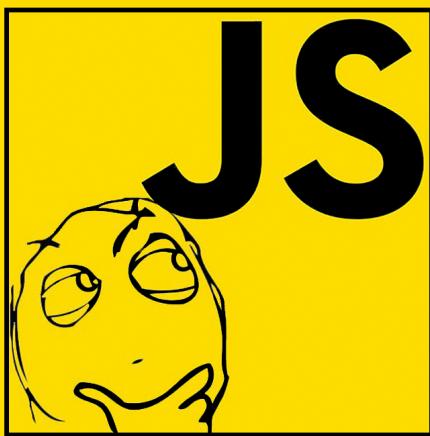
```
1 const cars = [
2     {id: 1, model: 'Ferrari'},
3     {id: 2, model: 'Audi'},
4     {id: 3, model: 'Toyota'},
5     {id: 4, model: 'Suzuki'}
6 ];
7
8 const filterCars = cars.filter(item => item.id % 2 === 0);
9
10 filterCars; // [{ id: 2, model: 'Audi' }, { id: 4, \
11 // model: 'Suzuki' }]
```

Как видишь, у нас получился массив только с теми элементами значения свойства `id`, которые являются четными.

Методы массивов: map, sort

JavaScript для тупых

Методы массивов map, sort



@javascriptstupid

Метод map

Метод `map` создает новый массив из элементов, которые будут являться результатами выполнения **callback-функции**.

У метода `map` имеется только один аргумент, который является **callback-функцией**. У этой функции имеется `3` аргумента, по аналогии с методами описанными ранее:

Первый аргумент – текущий элемент (элемент на текущей итерации).

Второй аргумент – индекс этого элемента в массиве.

Третий аргумент – сам массив.

Повторюсь, **callback-функция** может быть как **обычной** (`function declaration`), так и **стрелочной** (`arrow function`).

Пример использования:

```
1 const numbers = [2, 5, 7, 9];
2 const results = numbers.map(n => n ** 5);
3
4 results; // [32, 3125, 16807, 59049]
```

Из исходного массива `numbers`, с помощью метода `map` мы создали новый массив, в котором каждое число было возведено в `5-ю степень`.

Метод `sort`

Метод `sort`, как несложно догадаться по названию – **сортирует массив**. При этом, сортирует он его **на месте**, что означает изменение исходного массива. Плюсом к этому, данный метод еще и возвращает тот же самый массив, т.е. результат выполнения данного метода можно ещё и в другую переменную запихнуть (но это делается крайне редко).

Самый простой пример, который можно придумать, отсортировать числа, например, по возрастанию, но мы начнем не с них, а с буквок:

```
1 const characters = ['d', 'z', 'a', 'c', 'y'];
2 characters.sort();
3
4 characters; // ['a', 'c', 'd', 'y', 'z']
```

Как видишь, метод расположил буквы в соответствии с их расположением в латинском алфавите. Но что же будет, если применить метод `sort` к массиву с числами? Глянем:

```
1 const numbers = [45, 4, 2, 1, -50, 300, 0];
2 numbers.sort();
3
4 numbers; // [-50, 0, 1, 2, 300, 4, 45]
```

И казалось бы, что может пойти не так?! Но, всмотрись внимательно и увидишь, что значение `300` заняло далеко не своё место в этом танце =) Почему так?

ПотомуШТА =). А если серьезно, то случилось это по той причине, что метод `sort` по умолчанию при работе с элементами массива преобразует их (элементы) к строковому типу и для сортировки применяется **лексикографический порядок**. А если сравнивать строки, то строка `300` будет больше чем строка `4`. Чтобы такой чепухи не было, нам нужно задать свой порядок сортировки.

Ты же заметил, что мы не передавали никакого аргумента в метод `sort`, а они у него есть. И да, это снова **callback-функция**. Если ты ее передашь, то она будет (**должна**) задавать правила сортировки для массива.

Главное правило **callback-функции** для данного метода – для пары значений она должна возвращать:

- `1` (или любое положительное число) – если первое значение больше второго;
- `0` – если значения равны;
- `-1` (или любое отрицательное число) – если первое значение меньше второго.

Теперь подкрепим понимание кодом:

```
1 const numbers = [45, 4, 2, 1, -50, 300, 0];
2
3 function compare(a, b) {
4     if (a > b) return 1;
5     if (a === b) return 0;
6     if (a < b) return -1;
7 }
8
9 numbers.sort(compare);
10
11 numbers; // [-50, 0, 1, 2, 4, 45, 300]
```

Вот оно, чудо из всех чудес! Теперь сортировка отработала совершенно правильно.

Что же мы поменяли? Да в целом, ничего. Мы создали функцию, в которой описали при каких условиях и что возвращать, используя правило выше. И эту функцию мы передали в аргумент метода. И все заработало как часы.

Конкретно данный случай можно еще и упростить, если использовать стрелочную функцию и немножко включить логику. Смотри:

```
1 const numbers = [45, 4, 2, 1, -50, 300, 0];
2
3 numbers.sort((a, b) => a - b);
4
5 numbers; // [-50, 0, 1, 2, 4, 45, 300]
```

Воу! Код колоссально сократился, но чего же мы такого тут поменяли?

Во-первых, мы ввели стрелочную функцию, что автоматически в львиной доле случаев делает код короче.

Во-вторых, посмотри на эту функцию: `(a, b) => a - b`. Здесь так же фигурируют `a` и `b`, как и в примере выше и мы возвращаем их разницу. Зачем?

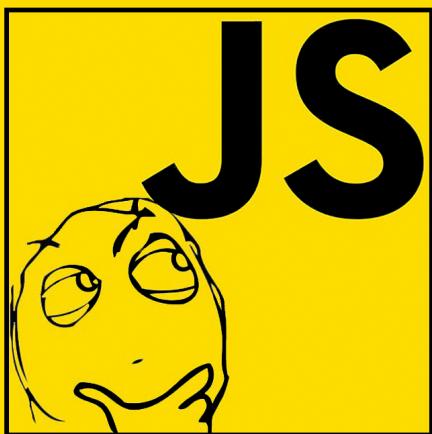
Ты не забыл о главном правиле? Функция должна вернуть `1` (или любое положительное число), `0` или `-1` (или любое отрицательное число).

Именно это здесь и вернется. Если числа будут равны, то результат `a - b` даст нам значение `0`. Если `a > b`, то нам вернется положительное число, если `a < b`, то вернётся отрицательное число. Таким образом, правило у нас исполнится и метод отработает как нужно.

Методы массивов: reverse, split, join

JavaScript для тупых

Методы массивов reverse, split, join



@javascriptstupid

Метод `reverse` попроще тех, что мы уже прошли ранее, отдохнувшись, передохни =). По названию, ты должен догадаться для чего он нужен.

Данный метод делает самую простую, но порой очень полезную штуку – переворачивает массив, т.е. **первый элемент становится последним, а последний – первым**. Метод отрабатывает **по месту**, что означает, повторюсь, **меняет исходный массив**.

```
1 const languages = ['python', 'java', 'c++', 'js'];
2 languages.reverse();
3
4 languages; // ['js', 'c++', 'java', 'python']
```

Здесь по сути и комментировать нечего, результат на лицо.

Метод `split`

Полезный метод `split`. Что же он делает? Он позволяет получить из строки – массив. Результат данного метода нужно куда-то записать, он не меняет исходный массив.

Сразу к примеру:

```
1 const words = 'cat, dog, snake';
2 const arr = words.split(',');
3
4 arr; // ['cat', ' dog', ' snake']
```

В метод `split` мы передали в аргументах строку, в которой содержится запятая . Надеюсь ты уже смекнул зачем. Фактически, аргумент, который мы передаем является разделителем , т.е. символом, который будет расчленять строку на элементы массива.

В данный метод, в качестве аргумента можно засунуть просто пустую строку. В таком случае, каждый символ будет представлен отдельным элементом массива:

```
1 const str = 'JavaScript';
2 const arr = str.split('');
3
4 arr; // ['J', 'a', 'v', 'a', 'S', 'c', 'r', 'i', 'p', 't']
```

Если же совсем не указывать аргумент, то вся строка будет помещена в массив:

```
1 const str = 'JavaScript';
2 const arr = str.split();
3
4 arr; // ['JavaScript']
```

Метод `join`

Метод `join` делает то же самое, что и метод `split`, но в обратную сторону — переводит массив в строку вставляя между элементами разделитель:

```
1 const arr = ['cat', 'dog', 'snake'];
2 const str = arr.join('#');
3
4 str; // cat#dog#snake
```

Если в качестве разделителя указать пустую строку, то весь массив замиксится в одну большую строку:

```
1 const arr = ['cat', 'dog', 'snake'];
2 const str = arr.join('');
3
4 str; // catdogsname
```

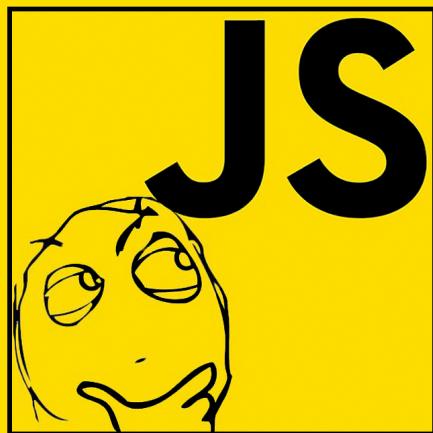
Если же не указывать аргумент вовсе, то, по умолчанию, метод разделит элементы запятыми:

```
1 const arr = ['cat', 'dog', 'snake'];
2 const str = arr.join();
3
4 str; // cat,dog,snake
```

Методы массивов: indexOf, lastIndexOf, includes

JavaScript для тупых

Методы массивов indexOf, lastIndexOf, includes



@javascriptstupid

Метод indexOf

Метод `indexOf` позволяет найти индекс элемента в массиве по значению.

```
1 const languages = ['python', 'java', 'js'];
2 const javaIndex = languages.indexOf('java');
3
4 console.log(javaIndex); // 1
```

Метод принимает **2 аргумента**, второй из которых является необязательным.

Первый аргумент – значение, которое ты пытаешься найти в массиве.

Второй аргумент – позволяет указать с какого индекса начинать поиск.

В примере выше я искал индекс значения `java` в массиве `languages`. Результатом выполнения метода стало значение `1`. Т.е. метод отработал и выдал индекс.

Но, если в массиве не существует указанного тобой значения, то метод выдаст значение `-1`.

Метод `lastIndexOf`

Полностью аналогичен с методом `indexOf` за тем исключением, что начинает искать указанное значение с конца массива. Используется достаточно редко, но знать о нем нужно, так как в некоторых случаях этот метод может быть более оптимизированным выбором для скорости выполнения кода.

Метод `includes`

Метод `includes` проверяет, существует ли указанное значение в массиве.

Возвращает `true` в том случае если существует и `false`, если значение отсутствует.

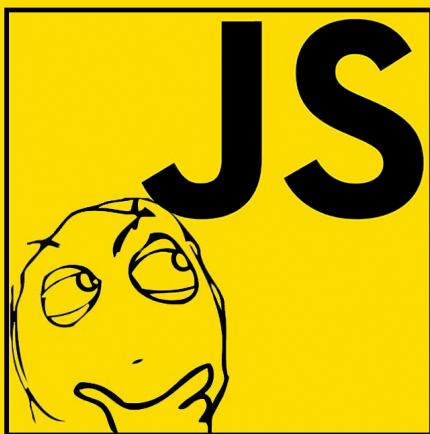
```
1 const languages = ['python', 'java', 'js'];
2
3 console.log(languages.includes('js')); // true
4 console.log(languages.includes('c++')) // false
```

Метод, по аналогии с `indexOf`, имеет второй аргумент, которым можно указать с какого индекса начинать поиск.

Методы массивов: reduce, spread

JavaScript для тупых

Методы массивов reduce, spread



@javascriptstupid

Метод `reduce`

Метод `reduce`, пожалуй, самый сложный метод при работе с массивом, поэтому придется напрячь свои извилины и внимательно вникнуть в суть происходящего.

Метод работает с элементами **слева направо**.

Один из главных аргументов метода, как и у многих других – **callback-функция**. Данный метод необычен тем, что результат **callback-функции** сохраняется для следующего элемента. Т.е. при вызове **callback** на следующем элементе, метод позволяет получить доступ к значению вернувшемуся из этого же **callback**, но на предыдущем элементе. Грубо говоря, метод выполнил **callback-функцию** на первом элементе массива и переходя ко второму элементу, мы имеем доступ к значению которое вернул **callback** на первом элементе. Итак, пока ты не запутался и не упал на пол, нужно добавить в объяснения щепотку кода.

Давай с помощью данного метода посчитаем сумму всех элементов массива:

```
1 const numbers = [50, 50, 1];
2
3 const results = numbers.reduce(function(previousValue, \
4 currentElem){
5     return previousValue + currentElem;
6 }, 0);
7
8 results; // 101
```

Пожалуй, стоит объяснить, что здесь происходит.

Итак, разберем все аргументы, которые может принимать **callback-функция**:

- Первый аргумент (`previousValue`) – значение, которое вернула **callback-функция** для предыдущего элемента;
- Второй аргумент (`currentElem`) – текущий элемент;
- Третий аргумент – индекс текущего элемента в массиве;
- Четвертый аргумент – сам массив, по которому бежит метод `reduce`.

Мы в своей **callback-функции** использовали только первые два аргумента, т.к. для нашей задачи этого достаточно.

Наша **callback-функция** возвращает следующее значение:

```
return previousValue + currentElem;
```

Т.е. каждый новый элемент суммируется с предыдущим. По итогу мы и получим сумму всех элементов.

Второй аргумент метода `reduce` – начальное значение (`initialValue`) для всего метода. В нашем случае мы указали значение `0`. Если бы мы указали, например, значение `100`, то результатом работы метода стало бы значение не `101`, а `201`. Т.е., если тебе нужно к какому-то числу прибавить сумму всего массива, то это самое число нужно указать в качестве начального значения.

У данного метода есть брат акробат, который выполняет все то же самое и работает точно так же, но начинает пробегать по элементам **справа налево** и называется данный метод – `reduceRight`.

Оператор spread

Оператор `spread` не является методом массива, но достаточно полезен при работе с массивами, поэтому пройти мимо него – преступление.

Оператор выглядит достаточно непривычно относительно других ранее увиденных тобой операторов. Выглядит он так: `...`



Да-да, этот оператор представляет собой три точки =)

С его помощью, можно **объединять массивы**:

```
1 const numbers = [3, 4, 5];
2 const arr = [1, 2, ...numbers];
3
4 arr; // [1, 2, 3, 4, 5]
```

Надеюсь ты заметил, что внутри определения нового массива, одним из элементов я указал другой массив с оператором `spread` перед ним. Тем самым мы **объединяем 2 массива в один**. Но можно не 2, а сколько угодно:

```
1 const numbers = [1, 2, 3];
2 const otherNumbers = [4, 5, 6];
3
4 const resultArr = [...numbers, ...otherNumbers];
5
6 resultArr; // [1, 2, 3, 4, 5, 6]
```

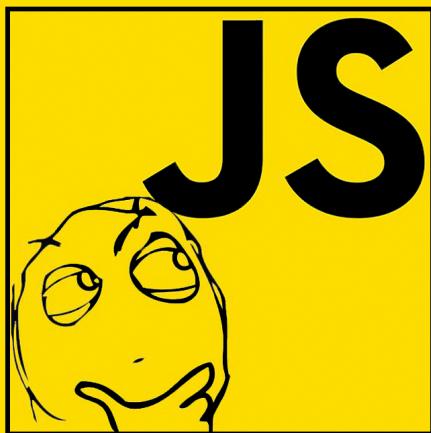
Так же с помощью данного оператора можно скопировать массив . Но запомни, что таким методом *можно копировать только массивы с примитивными значениями* (числа, строки, булевые значения), но никак не массивы, элементами которых являются **сложные объекты**. Пример копирования:

```
1 const numbers = [1, 2, 3, 4, 5];
2 const copyNumbers = [...numbers];
3
4 copyNumbers; // [1, 2, 3, 4, 5]
```

Методы массивов: Array.isArray, some, every

JavaScript для тупых

Методы массивов Array.isArray, some, every



@javascriptstupid

Метод Array.isArray

Порой требуется понять, являются ли данные, которые, например, пришли с сервера – массивом . Как мы можем это узнать? Существует оператор `typeof` , но в **JavaScript** нет отдельного типа для массивов. **JavaScript** не различает массив от объекта: для него и то и другое – `объекты` . Но задачу как-то надо решать, именно для этого и существует статический метод `isArray` в объекте `Array` .

```
1 const obj = {name: 'Толян'};  
2 const arr = [1, 2, 3];  
3  
4 Array.isArray(arr); // true  
5 Array.isArray(obj); // false
```

Метод some

Представим, что у тебя есть массив чисел. При этом, предположим, что на момент работы с массивом ты точно не знаешь, какими числами он заполнен.

И тебе нужно узнать, что в массиве есть хотя бы один элемент, который можно разделить на 2 и полученное значение будет больше, либо равно 10 .

В голову может прийти множество вариантов решений. Но метод some подойдет здесь намного лучше:

```
1 const numbers = [20, 45, 12, -4, 5, 7];
2
3 const has = numbers.some(n => (n/2) >= 10);
4
5 has; // true
```

В данном случае под наше условие подойдет даже 2 числа: 20 и 45 , поэтому метод выдаст нам true .

Метод every

Метод every похож на метод some . Но, по названию должно быть понятно, что метод every вернет true только в том случае, если все элементы массива подходят под заданное условие.

Давай проверим каждый элемент массива на условие: каждый элемент делится на 2 и полученное значение больше или равно 10 .

```
1 const numbers = [20, 30, 40];
2 const otherNumbers = [20, 10, 40];
3
4 const hasInNumbers = numbers.every(n => (n/2) >= 10);
5 const hasInOtherNumbers = otherNumbers.every(n => (n/2) \
6 >= 10);
7
8 hasInNumbers; // true
```

```
9 hasInOtherNumbers; // false
```

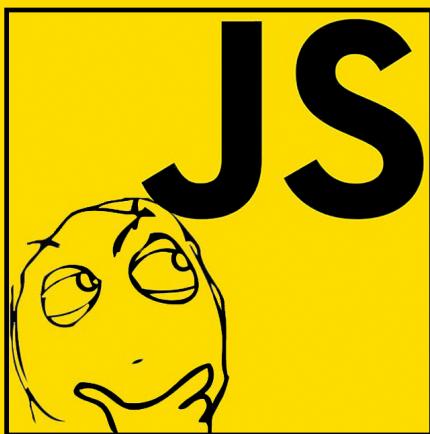
Все элементы массива `numbers` подходят под условие – поэтому метод `every` выдает нам значение `true`.

А вот в массиве `otherNumbers` под условие не подходит значение `10`, поэтому `every` вернет – `false`.

Методы массивов: flat, fill

JavaScript для тупых

Методы массивов flat, fill



@javascriptstupid

Метод flat

Данный метод появился недавно, но он очень крут. Он создает новый массив из всех подмассивов содержащихся в нем уменьшая мерность на указанное значение (по умолчанию 1). Наверное звучит очень сложно и запутанно и, скорее всего, прямо сейчас ты мало где сможешь применить данный метод на практике. Но пример все равно давай посмотрим:

```
1 const arr = [1, 2, 3, [4, 5], [[7,8]]];  
2 const transformArr = arr.flat();  
3  
4 transformArr; // [1, 2, 3, 4, 5, [7, 8]]
```

Как видишь, массив arr был преобразован в новый. Как ты можешь заметить, все подмассивы, которые были внутри массива arr как бы **распаковались** на 1

уровень вложенности. Если третьим элементом в массиве `arr` был массив `[4, 5]`, то сейчас он распаковался и перестал существовать как массив.

В этом же массиве `arr` 4-м элементом был массив с массивом (т.е. это 2 уровня вложенности), но теперь он выглядит как простой массив `[7, 8]`, т.е. начальный массив с массивом распаковался в обычный массив (один уровень вложенности исчез).

Данный уровень **сглаживания** вложенности, который, повторюсь, по умолчанию равен `1` – может быть задан. Для этого в метод `flat` нужно передать число, которое и будет являться этим уровнем сглаживания.

Так как мы не всегда можем знать уровень вложенности подмассивов, мы можем указать в качестве параметра значение бесконечности – `Infinity`. Это позволит нам убрать абсолютно все вложенности во всех подмассивах. Пример:

```
1 const arr = [1, 2, 3, [4, 5], [[7,8]]];
2 const transformArr = arr.flat(Infinity);
3
4 transformArr; // [1, 2, 3, 4, 5, 7, 8]
```

Какие бы глубины вложенности не были теперь массивы внутри массива `arr`, они всегда будут распакованы в один одноуровневый массив.

Метод `fill`

Метод `fill` заменяет или заполняет элементы массива одним и тем же значением. Для замены или заполнения конкретных отрезков массива можно указать индексы начального и конечного элементов, которые требуют заполнения/замены.

```
1 const arr = [1, 2, 3, 4];
2 arr.fill(0, 2, 4)
3
4 arr; // [1, 2, 0, 0]
```

Рассмотрим вызов метода:

```
arr.fill(0, 2, 4)
```

Первый аргумент – значение, которое заполняет/заменяет элементы массива;

Второй аргумент – индекс после которого необходимо вставить указанное значение;

Третий аргумент – индекс до которого необходимо вставить указанное значение.

Получаем:

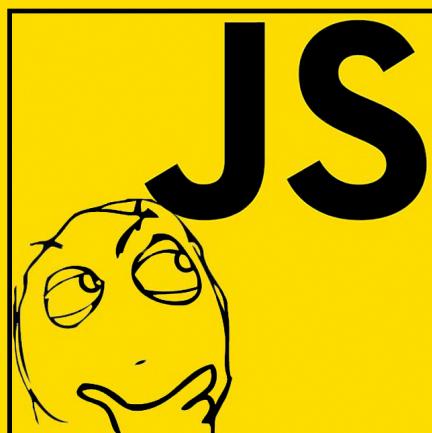
```
arr; // [1, 2, 0, 0]
```

Что полностью соответствует ожиданиям.

Планирование вызова функции: setTimeout

JavaScript для тупых

Планирование вызова функции: setTimeout



@javascriptstupid

Представь, у тебя есть задача вывести приветственное сообщение пользователю через 5 секунд после того, как он перешел на страницу. Как это организовать средствами JavaScript?

В этом вопросе нам поможет метод объекта `window` – `setTimeout()`. Данный метод запускает через указанное в аргументах время, указанную в аргументах функцию. Пример использования:

```
1 // 1-й вариант использования
2 window.setTimeout(func, delay);
3
4 // 2-й вариант использования
5 setTimeout(func, delay);
```

Как видишь, данный метод объекта `window` можно вызывать и как обычную функцию (*2-й вариант использования*). Я, кстати, рекомендую использовать именно **2-й вариант**, так как он чаще всего встречается и используется.

В качестве аргументов мы передаем `func` и `delay`, где `func` – это функция, либо ссылка на функцию, которую должен запустить `setTimeout` после того, как пройдет время `delay` (указывается в миллисекундах). **Кстати, такие функции, которые передаются в другие функции в качестве аргументов – называются callback-функциями или функциями обратного вызова.**

Теперь вернемся к нашей задаче поставленной в начале урока и поприветствуем нашего пользователя:

```
1 function hello() {  
2     alert('Привет!');  
3 }  
4  
5 setTimeout(hello, 5000);
```

Функция `hello`, в данном примере, является **callback-функцией**, которую нужно запустить после того как пройдет 5000 миллисекунд (5 сек). В данном примере, в качестве первого аргумента, мы передали ссылку на функцию `hello`. Ссылка на функцию – это ее имя, а не вызов. Т.е.:

```
1 hello; // – это ссылка на функцию  
2 hello(); // – это вызов функции
```

В качестве аргумента мы должны передавать либо ссылку на функцию, либо описание функции (смотри ниже), но никогда не должны передавать вызов функции!

Есть и другие варианты передачи **callback-функции**, которые делают ровно тоже самое.

В качестве первого аргумента, мы можем передать **анонимную** (*безымянную*) функцию:

```
setTimeout(function() { alert('Привет!'); }, 5000);
```

Так же можем передать **анонимную стрелочную** функцию:

```
setTimeout(() => alert('Привет!'), 5000);
```

Небольшое отступление: использование **анонимных функций** позволяет не описывать **callback-функцию** в глобальной области видимости, т.е. **анонимная функция**, описание которой ты передашь в качестве первого аргумента будет доступна только для `setTimeout`, и не доступна ни для кого извне, что, иногда, является лучшим решением.

Продолжим.

Самое интересное, что мы можем передать строку, вместо функции и она выполнится:

```
setTimeout(`alert('Привет!')`, 5000);
```

Все варианты выполняют одну и ту же задачу, но, последний вариант с передачей строки не рекомендуется использовать так как это и стилистически некрасиво и в плане безопасности – **это плохой вариант**.

```
setTimeout(`alert('Привет!')`, 5000);
```

Стоит запомнить, что такой вариант может встретиться в чужом коде, но **в своём** **такой писать не стоит.**

Последнее что хотелось бы отметить, так это то, что метод `setTimeout` нужно **удалять**, чтобы не произошло **утечки памяти**. Делается это достаточно просто.

`setTimeout` на самом деле еще и возвращает данные, а именно, он возвращает свой идентификатор . Стоит понимать, что у каждого `setTimeout` этот идентификатор **индивидуальный, свой**. Чтобы не произошло утечки памяти, нужно удалять `setTimeout` после выполнения. Сделать это можно с помощью `clearTimeout(id)` , где в качестве аргумента нужно передать идентификатор вашего `setTimeout` .

Для начала давай сохраним идентификатор нашего `setTimeout` в константе `timerId` :

```
1 function hello() {  
2     alert('Привет');  
3 }  
4  
5 const timerId = setTimeout(hello, 5000);
```

Удалить `setTimeout` из памяти нам нужно после выполнения приветствия, поэтому добавим `clearInterval` в функцию `hello` , после вывода приветствия:

```
1 function hello() {  
2     alert('Привет');  
3     clearInterval(timerId);  
4 }  
5  
6 const timerId = setTimeout(hello, 5000);
```

Всё, теперь после выполнения приветствия `setTimeout` будет удален из памяти.

Еще один небольшой пример:

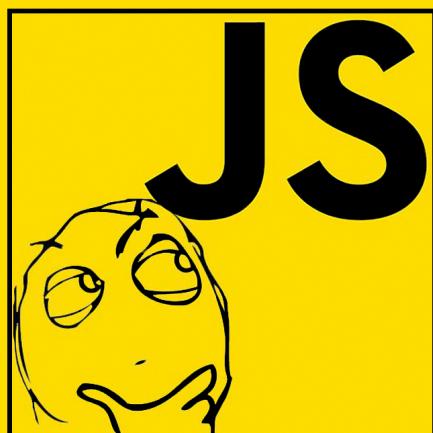
```
1 function hello() {  
2     alert('Привет');  
3 }  
4  
5 const timerId = setTimeout(hello, 5000);  
6 clearInterval(timerId);
```

В данном случае `setTimeout` не будет выполнен совсем, потому что мы вызвали `clearInterval` сразу же после объявления `setTimeout`, данным действием мы как бы **отменили (удалили) `setTimeout`**.

Планирование вызова функции: setInterval

JavaScript для тупых

Планирование вызова функции: setInterval



@javascriptstupid

Рассмотрим ещё один метод для планирования вызова функции – `setInterval`.

Принцип работы точно такой же, как и у его брата `setTimeout` (`setTimeout`), за одним небольшим исключением: `setInterval` повторяет вызов переданной тобою функции через указанное время.

Итак, как и в случае с `setTimeout` (лучше прочитай о `setTimeout`, будет намного понятнее), `setInterval` принимает **первым аргументом** либо **ссылку на функцию**, либо **функцию**, а **вторым аргументом** передается **время в миллисекундах**, через которое периодически будет выполняться переданная функция:

```
1 function hello() {  
2     console.log('Привет');  
3 }
```

```
4
5  setInterval(hello, 3000);
6
7 // каждые 3 секунды в консоль будет выводиться сообщение: \
8 // 'Привет'
```

Как только потребность в нём пропадает, данный метод лучше всего останавливать.

`setInterval`, как и `setTimeout`, возвращает идентификатор, по которому в дальнейшем можно остановить выполнение `setInterval`:

```
1 function hello() {
2   console.log('Привет');
3 }
4
5 const intervalId = setInterval(hello, 3000);
6
7 // команда ниже остановит выполнение setInterval
8 clearInterval(intervalId);
```

Давай попробуем выполнить следующую задачу: с помощью `setInterval` выведем в консоль слово "Привет" 3 раза. Каждое слово должно появляться через секунду после предыдущего.

Код:

```
1 let counter = 0;
2
3 function hello() {
4   console.log('Привет');
5
6   counter += 1;
7
8   if (counter === 3) {
9     clearInterval(intervalId);
10  }
11 }
12
```

```
13 const intervalId = setInterval(hello, 1000);
```

Итак, мы определили функцию `hello`, которую затем передали в качестве первого аргумента в метод `setInterval`, вторым аргументом указали время `1000` мс (`1` секунда) через которое постоянно нужно запускать функцию `hello`.

Также мы определили переменную `counter` (счётчик), которая будет считать количество запусков функции `hello` (напомню, по условию нам нужно запустить эту функцию только `3` раза).

В самой функции `hello`:

1. Сначала выводим в консоль слово "Привет";
2. Затем увеличиваем значение счётчика;
3. Строгим сравнением проверяем не равен ли счётчик значению `3`. Если условие выполняется, следовательно, слово "Привет" будет выведено `3` раза и далее нам нужно остановить выполнение `setInterval`, для этого нам поможет метод `clearInterval`.

Результат:

```
> let counter = 0;

function hello() {
    console.log('Привет');

    counter += 1;

    if (counter === 3) {
        clearInterval(intervalId);
    }
}

const intervalId = setInterval(hello, 1000);
< undefined

3 Привет
```

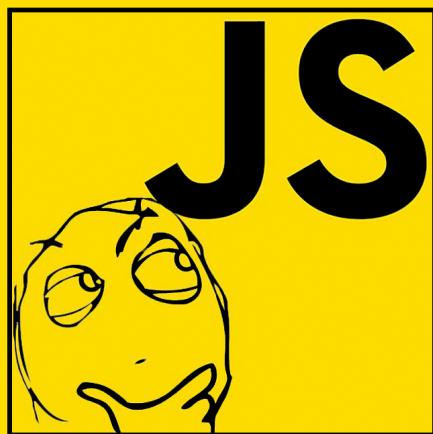
Перед словом "Привет" стоит цифра 3, это означает, что данная надпись трижды выводилась в консоль (так браузер сворачивает одинаковые значения выводимые в консоль, чтобы уменьшить количество мусора).

Собственно и всё. Задача выполнена. **Обязательно повтори ее на практике.**

Объекты. Свойства.

JavaScript для тупых

Объекты: свойства



@javascriptstupid

В предыдущих уроках мы рассматривали **примитивные типы** (исключением являются **массивы**). Эти типы важны и нужны, но так как язык **JavaScript является объектно-ориентированным (ООП)**, и в нём часто используются **ООП подходы**, то не знать об объектах и не уметь ими пользоваться – огромный пробел в знаниях.

Объект – это сложная сущность, которая может хранить в себе целый набор различных данных, состоящих из **примитивных значений**, **массивов** и **функций**.

Как создать объект?

Есть несколько способов создать переменную, в которой будет храниться объект. Если нам нужно создать пустой (чистый) объект, то есть **2 варианта**:

1. `const cat = new Object();`
2. `const cat = {};`

Синтаксис 1-го варианта называется **конструктором объекта**, 2-го – **литералом объекта** (или **литеральной нотацией**) .

Каким вариантом пользоваться? Чаще всего в реальных проектах ты сможешь увидеть 2-й вариант, т.е. `const cat = {};`

Какие же преимущества есть у объекта? Представим, что мы желаем записать информацию о твоём котике Ваське. К примеру нам нужна следующая информация: **кличка , возраст , цвет и вес .**

Как бы ты сейчас стал записывать в коде эти данные? Скорее всего, примерно так:

```
1 let catName = 'Вася';
2 let catAge = 2;
3 let catColor = 'рыжий';
4 let catWeight = 3;
```

Для того, чтобы записать информацию о котике, мы создали целых **4 переменных**. А представь, если тебе нужно будет заполнить данные о каком-то сложном объекте, к примеру об автомобиле, который имеет огромную гору различных характеристик. Создавать 100500 переменных – **это очень плохо**, как минимум потому, что ты запутаешься в этих переменных очень быстро.

Если ты думаешь, что не запутаешься, то представь, что тебе нужно заполнить информацию о трёх автомобилях, когда для каждого нужно прописать по **50** характеристик! Таким методом, на **3** автомобиля у тебя получится **150** переменных.

А теперь давай представим, что у тебя есть возможность вместо **150** переменных использовать только **3**. Как тебе такой вариант? Я думаю, что куда лучше и проще.

Вернемся к котику. Создаем объект для нашего котика:

```
1 const catVasya = {
```

```
2     name: 'Вася',
3     age: 2,
4     color: 'рыжий',
5     weight: 3
6 };
```

Вуаля, вместо 4-х переменных получили всего одну с именем `cat`. Теперь для того, чтобы вытащить какое-то значение из объекта, нужно всего лишь обратится к нужному полю в объекте. Например, мы хотим получить **цвет котика** из этого объекта :

```
catVasya.color;
```

Нужно написать название объекта и через точку указать нужное поле.

Свойства объекта

На самом деле то, что выше я называл полями в объекте называется **свойствами**. В свойстве могут храниться любые **примитивные данные** (строки , числа , булевые значения) и другие объекты (**массивы в JavaScript** – тоже имеют тип `Object`).

Свойства определяются по данному сценарию: внутри **фигурных скобок** `{...}` , мы должны сначала написать **название свойства** и, затем, через двоеточие : указать **значение этого свойства**:

```
1 {
2     имяСвойства: 'значение свойства',
3     имяСвойства2: 5,
4     имяСвойства3: true,
5     имяСвойства4: ['а', 'б', 'в'],
6     имяСвойства5: {
7         // здесь указать свойства другого объекта
8     }
9 }
```

Как видишь, мы можем записать в свойство объекта любой валидный тип данных, хоть примитивы, хоть массивы, хоть другие объекты.

Свойства в обязательном порядке разделяются друг от друга запятой.

Хочу обратить твоё внимание на то, что **имя свойства внутри объекта** можно определять как в **кавычках**, так и **без них**.

Взгляни на эти примеры:

```
const catVasya = {  
    name: 'Вася',  
    age: 2,  
    color: 'рыжий',  
    weight: 3  
};
```

Пример № 1

```
const catVasya = {  
    'name': 'Вася',  
    'age': 2,  
    'color': 'рыжий',  
    'weight': 3  
};
```

Пример № 2

В примерах приведён один и тот же объект, но с разным подходом в определении имени свойств. Какой правильный?

В данном случае, правильным будет использование объекта, где свойства определены **без кавычек**:

```
const catVasya = {  
    name: 'Вася',  
    age: 2,  
    color: 'рыжий',  
    weight: 3  
};
```

Почему? Все просто. Кавычки стоит использовать только тогда, когда ты хочешь задать **необычное** название для свойства. Например:

```
const catVasya = {  
    name: 'Вася',  
    age: 2,  
    color: 'рыжий',  
    weight: 3,  
    'other prop': 'bla-bla-bla'  
};
```

Я добавил котику свойство с именем `'other prop'`, при этом я отошёл от **хороших практик JavaScript** и не использовал стиль `camelCase`, а просто разделил эти слова **пробелом**. Самое интересное, что не будет никакой ошибки, если такое имя свойства

указать в кавычках. Но, если их убрать, то, конечно же, все сломается и объект будет определен некорректно.

При этом, если сейчас обратиться к новому свойству через символ точки:

```
catVasya.other prop;
```

Мы получим ошибку по той простой причине, что **JavaScript** просто не поймёт, что ты хотел обратиться к этому свойству. Потому что для **JavaScript** это будет выглядеть как то, что ты пытаешься обратиться к свойству объекта `catVasya` с именем `other`, затем ставишь пробел и пытаешься обратиться к переменной `prop`, которая не определена.

Поэтому, для того, чтобы обратиться к такому свойству нужно использовать следующий синтаксис:

```
catVasya['other prop'];
```

Данный случай разобран только для примера. Никогда, слышишь!? Никогда не используй в названии свойств объекта символ пробела! Это очень-очень плохо!

Вычисляемые свойства

В объекте можно определить свойство, имя которого заранее неизвестно. Звучит страшно и непонятно, да? Но сейчас поясню.

Представь, что имя свойства заранее неизвестно по той причине, что оно откуда-то приходит позже, либо его вводит пользователь, но нам все-равно нужно его как-то добавить. Объекты в **JavaScript** позволяют это сделать.

Давай будем просить пользователя вводить имя свойства и его значение:

```
1 let propName = prompt('Введи название свойства', \  
2 'tailLength');  
3 let propValue = prompt('Введи значение свойства', 20);
```

Мы завели переменные с именами `propName` и `propValue`. Оператор `prompt` спросит у пользователя как назвать свойство и какое значение в него положить.

По умолчанию (если пользователь ничего не укажет), мы запишем свойство `tailLength`, что означает - длина хвоста и передадим в него значение `20`.

Давай попробуем добавить данное свойство в объект. Для этого нужно при определении имени свойства обернуть переменную `propName` в квадратные скобки и тогда значение этой переменной будет указано как имя свойства объекта, а в само значение свойства нам нужно просто записать переменную `propValue`:

```
let propName = prompt('Введи название свойства', 'tailLength');  
let propValue = prompt('Введи значение свойства', 20);  
  
const catVasya = {  
    name: 'Вася',  
    age: 2,  
    color: 'рыжий',  
    weight: 3,  
    [propName]: propValue  
};  
  
console.log(catVasya);  
▼ {name: "Вася", age: 2, color: "рыжий", weight: 3, tailLength: "20"}  
  name: "Вася"  
  age: 2  
  color: "рыжий"  
  weight: 3  
  tailLength: "20"
```

Надеюсь этот пример наглядно показывает, что добавить свойство с заранее неизвестным именем не так уж и сложно, как может показаться на первый взгляд.

Именование свойств объекта

Важным аспектом является **именование свойств объекта**. Есть правило, которое гласит о том, что **имена свойств должны быть указаны как имена существительные**. В нашем примере это: `name` – имя, `age` – возраст, `color` – цвет, `weight` – вес.

Здесь очень просто провести аналогию. **Свойство любого реального объекта** – это какое-либо описание этого объекта: *цвет, возраст, рост, запах, вкус и т.д.* Все это и всегда является **именами существительными**.

Но, ведь у объекта могут быть **способности**: *передвигаться, говорить, летать и производить какие-то другие действия*. **Как быть с действиями?**

Действия любых объектов, в переводе на язык программирования – это **функции**.

Но, о функциях внутри объекта – в следующем уроке.

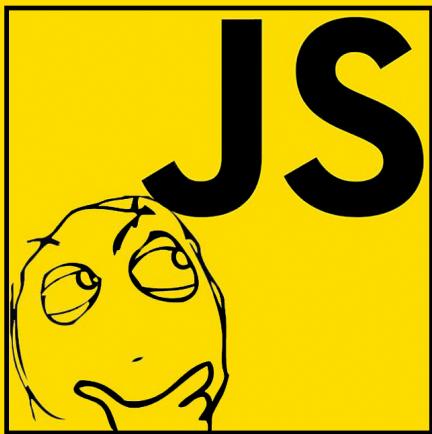
Домашнее задание

1. Создать объект **игрушка**
2. Добавить этому объекту свойства **имя и материал**
3. Добавить в объект **вычисляемое свойство**, в котором будет содержаться **цена на игрушку**.

Объекты. Методы

JavaScript для тупых

Объекты: методы



@javascriptstupid

Что такое метод объекта?

На самом деле **метод** – это **обычная функция**. Ничего сверхъестественного. Обычна, особо ничем не примечательна **функция**, которая просто-напросто находится внутри объекта и как-то связана с самим этим объектом.

Добавление метода

В прошлом [уроке](#) мы рассматривали свойства объекта и создали объект котика:

```
1 const catVasya = {  
2     name: 'Вася',  
3     age: 2,  
4     color: 'рыжий',  
5     weight: 3  
6 };
```

Так вот, мы имеем **имя котика**, его **возраст, цвет и вес**. Получается, мы дали котику некое описание, но вот делать наш котик пока что ничего не умеет.

Чтобы он что-то научился делать нам нужно добавить в **объект метод (функцию)**. Давай научим котика мяукать, для этого добавим в объект метод под названием mew:

```
1 const catVasya = {  
2     name: 'Вася',  
3     age: 2,  
4     color: 'рыжий',  
5     weight: 3,  
6     mew: function() {  
7         console.log('мяу-мяу');  
8     }  
9 };
```

Как видишь, функция в объект добавляется точно так же, как и обычное свойство, за тем исключением, что значение этого свойства – **функция**.

Правда, хочу заметить, что это не единственный способ определять метод внутри объекта. Очень часто ты можешь увидеть такую запись:

```
1 mew() {  
2     console.log('мяу-мяу');  
3 }
```

Данная запись считается **краткой и используется достаточно часто**.

Вызов метода

Теперь у получившегося объекта мы можем вызвать данный метод, как обычную функцию:

```
1 catVasya.mew();
2 // выведет в консоль: 'мяу-мяу'
```

В целом, все очень просто. Но, согласись, если бы на этом все заканчивалось, то особого смысла в методах и не было бы. Но, есть одна очень ценная вещь, которую умеют методы – **обращаться к другим свойствам и методам того же объекта**.

Давай представим, что мы попали в сказочную страну, где животные разговаривают. Следовательно, наш котик, может назвать своё имя. Поэтому, добавим ему такую возможность, добавив в наш объект **метод sayName**.

Внутри объекта лежит свойство `name`, следовательно, из метода `sayName` нам как-то нужно **достучаться** до этого свойства. Делается это просто, с помощью **ключевого свойства this**:

```
1 const catVasya = {
2     name: 'Вася',
3     age: 2,
4     color: 'рыжий',
5     weight: 3,
6     mew: function() {
7         console.log('мяу-мяу');
8     },
9     sayName() {
10         console.log('Привет, я кот ' + this.name);
11     }
12 };
13
14 catVasya.sayName();
15 // выведет в консоль: 'Привет, я кот Вася'
```

Здесь я специально использовал другой вариант определения метода `sayName`, для того, чтобы показать, что в нем нет ничего страшного.

Внутри этого метода мы вызываем `this.name`, что фактически означает: **возьми свойство name из текущего объекта**. С помощью `this` мы можем обращаться к любым свойствам и к любым методам текущего объекта.

Именование методов

Хотелось бы заострить твоё внимание на одном важном аспекте – **именование методов**.

Если **свойства** мы всегда называем **именем существительным**, то **методы** мы всегда называем используя **глаголы**. В нашем примере оба добавленных метода имеют глагол в своем названии: `mew` – мяукать, `sayName` – назвать имя.

Всегда называй методы с использованием глаголов, таковы правила и это поможет различать методы от свойств.

Домашнее задание

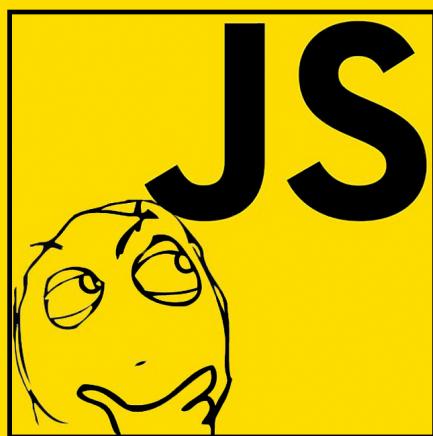
Добавить несколько **методов** нашему получившемуся котику, с помощью которых, он мог бы рассказать о своем **весе и цвете**.

Эти **методы** определить разными способами, которые мы прошли с тобой выше.

this, call, apply, bind

JavaScript для тупых

this, call, apply, bind



@javascriptstupid

В этом уроке расскажу о таких надуманно сложных вещах, как `call`, `bind`, `apply`, но и, соответственно, затрону `this`. Все эти слова связаны одним словом – **контекст**. Но, обо всем по порядку.

Многие боятся даже подходить к этой теме, потому как им кажется, что это что-то невероятно сложная и непонятная тема для новичков. Поэтому, надеюсь, что сегодня я развенчуя все ваши страхи и непонятки по этой теме.

Все примеры будут сделаны на основе **функций**, поэтому нужно представлять как они работают.

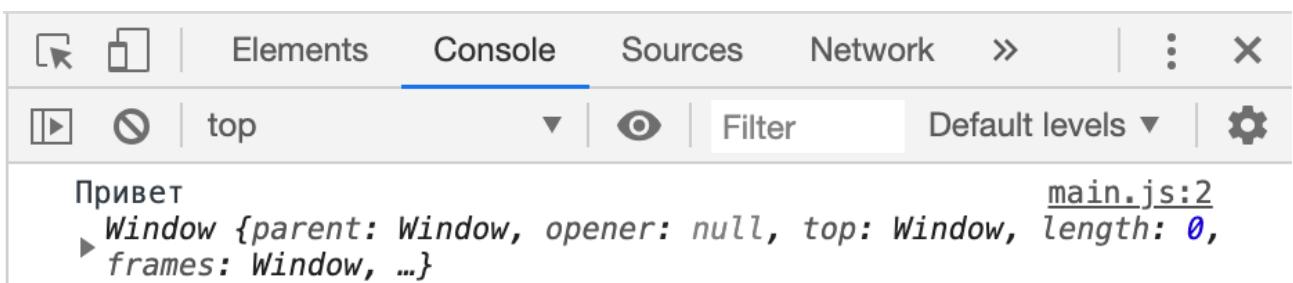
Предыдущие уроки по функциям: [Функции](#), [Функции. Возврат значения](#)

Ключевое слово this

Создадим обычную **функцию** и сразу же вызовем её:

```
1 function hi() {  
2     console.log('Привет', this);  
3 }  
4  
5 hi();
```

Функция выводит в консоль слово Привет и this. Если заглянуть в консоль, то увидим следующее:



Если со словом Привет все понятно, то вот что это такое вывелоось на месте this ?
Почему вывелся какой-то объект Window ?

Когда мы создаем глобальные объекты, то они автоматически начинают принадлежать глобальному объекту window – самому главному объекту в браузере.

На самом деле все методы, которые ты часто вызываешь, например:

console.log() , alert , prompt и т.д. находятся внутри объекта window и являются его **методами**.

Это достаточно просто проверить. Просто попробуй вызвать эти методы следующим образом:

- window.console.log('Привет');
- window.alert('Привет');
- window.prompt('Как тебя зовут?');

Надеюсь ты протестировал и понял, что абсолютно ничего не изменилось. А теперь, вернемся к нашей функции:

```
1 function hi() {  
2     console.log('Привет', this);  
3 }  
4  
5 hi();
```

На самом деле, когда мы вызываем функцию, то мы вызываем её так же из объекта `window`, т.е. вызов нашей функции можно переписать так:

```
window.hi();
```

Из всего выше сказанного можно сделать вывод, что наша **функция** запускается из **объекта** `window`, т.е. **контекст**, в котором выполняется наша **функция** так же равна **объекту** `window`. А ключевое слово `this` как раз и содержит внутри себя **контекст**, в котором вызывается **функция/метод**.

А теперь создадим **объект**:

```
1 const man = {  
2     name: 'Вася',  
3     lastName: 'Пупкин',  
4     age: 30,  
5     sayHi: hi  
6 };
```

Простой объект описывающий абстрактного мужчину. Обрати внимание на метод `sayHi`. В него мы передали ссылку на **функцию**, которую создали в самом начале.

Давай вызовем метод `sayHi` из нашего **объекта**:

```
man.sayHi();
```

А теперь посмотрим в консоль, что же теперь вывелоось на месте `this`. Получаем следующий результат:

```
Привет main.js:2
▶ {name: "Вася", lastName: "Пупкин", age: 30, sayHi: f}
```

> |

В этом случае значение `this` (**контекста**) становится равным самому **объекту**.

Почему так случилось? Все очень просто.

Первый раз мы вызывали **функцию** `hi` и **значение** `this`, внутри неё было равно **объекту** `Window`, потому что мы запускали **функцию в контексте объекта** `Window`.

А сейчас, мы создали свой объект, и запустили функцию `hi` из него. Получается, что место вызова функции изменилось с объекта `Window` на объект `man`.

Следовательно, изменилось и значение `this` с `Window` на `man`.

Теперь коротко: значение `this` равно тому объекту в контексте которого было вызвано.

С контекстом разобрались, давай разбираться с этими страшными методами:

`call`, `apply`, `bind`.

Метод `bind`

Мы ранее создали объект `man`, а теперь еще создадим объект `woman`. Итого, получим:

```
1 const man = {
2   name: 'Вася',
3   lastName: 'Пупкин',
4   age: 30,
5   sayHi: hi
6 };
7
8 const woman = {
```

```
9     name: 'Катя',
10    lastName: 'Иванова',
11    age: 25
12 };
```

Итак, мы имеем **2 объекта**, один из которых описывает *мужчину*, а второй *женщину*.

Для примера, создадим еще один объект (**логгер**), которому добавим один **метод**:

```
1 const Logger = {
2   info: function() {
3     console.log('Имя: ', this.name);
4     console.log('Фамилия: ', this.lastName);
5     console.log('Возраст: ', this.age);
6   }
7 };
```

Внутри объекта `Logger` мы создали метод `info`, который должен выводить информацию о наших объектах. Мы использовали ключевое слово `this` внутри этого метода:

```
1 console.log('Имя: ', this.name);
2 console.log('Фамилия: ', this.lastName);
3 console.log('Возраст: ', this.age);
```

Но что будет, если вызвать этот метод прямо сейчас?

```
Logger.info();
```

Результат:

Имя: undefined	main.js:19
Фамилия: undefined	main.js:20
Возраст: undefined	main.js:21

Мы получили `undefined` во всех случаях и это более чем логично. Т.к. мы обращаемся к ключевому слову `this` внутри метода `info`, который принадлежит объекту `Logger`, то и контекст, т.е. само ключевое `this` имеет значение, которое равно объекту `Logger`. А внутри этого объекта у нас имеется только один метод `info`. Никаких свойств с именами `name`, `lastName`, `age` у нас нет.

Как сделать так, чтобы с помощью объекта `Logger` и его метода `info` вывести данные, например, об объекте `man`?

Как раз здесь на помощь приходит тот самый страшный метод `bind`:

```
const loggerMan = Logger.info.bind(man);
```

У каждой функции в **JavaScript** существует метод `bind`, благодаря которому мы можем любой **функции** задать **контекст** внутри которого она должна будет выполняться.

Метод `bind` возвращает новую функцию к которой будет привязан тот контекст, который мы указали как аргумент метода `bind`. В нашем случае контекстом мы указали объект `man`. Так как `bind` возвращает новую функцию, то мы её запишем в константу `loggerMan`.

Теперь вызовем полученную функцию:

```
loggerMan();
```

Результат:

Имя:	Вася	main.js:19
Фамилия:	Пупкин	main.js:20
Возраст:	30	main.js:21

Как видишь, `bind` успешно **привязал** в качестве **контекста объект** `man` и информация о нём успешно вывелаась в консоль.

Теперь мы хотим получить информацию об объекте `woman` с помощью нашего логгера . Для этого делаем то же самое:

```
const loggerWoman = Logger.info.bind(woman);
```

С помощью `bind` мы **привязываем** новый **контекст** для **метода** `info` со **значением** `woman` .

Вызовем:

```
loggerWoman();
```

Результат:

Имя:	Катя	main.js:19
Фамилия:	Иванова	main.js:20
Возраст:	25	main.js:21

Вуаля, все так же прекрасно работает. И ведь совсем не сложно и не страшно, верно?

И последнее, что хотелось бы здесь отметить. В функцию `info` мы можем дополнителько передавать какие-либо параметры, если это нужно. Давай немнго исправим метод `info` в объекте `Logger` . Пускай он будет принимать **пол человека**:

```
1 const Logger = {
2   info: function(sex) {
3     console.log('Имя: ', this.name);
4     console.log('Фамилия: ', this.lastName);
5     console.log('Возраст: ', this.age);
6     console.log('Пол: ', sex);
7   }
8 };
```

Все что мы сделали это добавили в определение функции параметр `sex` и вывели его под всеми остальными данными. Остается вопрос, как нам туда это все передать, если нам еще и контекст нужно сменить. Все очень просто. У нас уже есть полученные функции для наших объектов:

```
1 const loggerMan = Logger.info.bind(man);
2 const loggerWoman = Logger.info.bind(woman);
```

Чтобы передать параметр `sex`, нам достаточно указать его аргументом наших функций при их вызове:

```
1 loggerMan('мужской');
2 loggerWoman('женский');
```

Смотрим на результат:

Имя: Вася

[main.js:19](#)

Фамилия: Пупкин

[main.js:20](#)

Возраст: 30

[main.js:21](#)

Пол: мужской

[main.js:22](#)

Имя:	Катя	main.js:19
Фамилия:	Иванова	main.js:20
Возраст:	25	main.js:21
Пол:	женский	main.js:22

Все прекрасно отработало. Но это не единственный способ передачи параметров.

Есть еще один способ передачи параметров – указать их при вызове метода `bind`.

Получим следующее:

```
1 const loggerMan = Logger.info.bind(man, 'мужской');
2 const loggerWoman = Logger.info.bind(woman, 'женский');
3
4 loggerMan();
5 loggerWoman();
```

Результат:

Имя:	Вася	main.js:19
Фамилия:	Пупкин	main.js:20
Возраст:	30	main.js:21
Пол:	мужской	main.js:22

Имя:	Катя	main.js:19
Фамилия:	Иванова	main.js:20
Возраст:	25	main.js:21
Пол:	женский	main.js:22

Результат никак не поменялся и все прекрасно продолжает работать.

Первый аргумент в методе `bind` – это сам **контекст**, который нужно **привязать к функции**. А вот дальше, через **запятую**, можно передавать сколько угодно

дополнительных параметров (аргументов), которые должны попасть в метод `info`. В нашем случае нам нужно было передать только один аргумент `sex`.

На этом с `bind` всё. Не такой уж и страшный зверь, если приглядеться.

Метод `call`

Что ж, продолжим убивать страх внутри тебя. Теперь на очереди метод `call`.

Ты не поверишь, но он делает то же самое, что и `bind`. С одной маленькой поправкой: если метод `bind` привязывает контекст и возвращает новую функцию с этим контекстом, то метод `call` привязывая контекст, сразу же вызывает указанную функцию, а не возвращает новую.

Продублирую наш код:

```
1 const Logger = {
2   info: function(sex) {
3     console.log('Имя: ', this.name);
4     console.log('Фамилия: ', this.lastName);
5     console.log('Возраст: ', this.age);
6     console.log('Пол: ', sex);
7   }
8 };
9
10 const man = {
11   name: 'Вася',
12   lastName: 'Пупкин',
13   age: 30,
14   sayHi: hi
15 };
16
17 const woman = {
18   name: 'Катя',
19   lastName: 'Иванова',
20   age: 25
21 };
22
23 const loggerMan = Logger.info.bind(man, 'мужской');
24 const loggerWoman = Logger.info.bind(woman, 'женский');
25
26 loggerMan();
```

```
27 loggerWoman();
```

Итак, давай **привяжем контекст** с помощью `call`, а не `bind` и посмотрим в чем же разница.

Было:

```
1 const loggerMan = Logger.info.bind(man, 'мужской');
2 const loggerWoman = Logger.info.bind(woman, 'женский');
```

Стало:

```
1 Logger.info.call(man, 'мужской');
2 Logger.info.call(woman, 'женский');
```

Так как метод `call` после привязки контекста сразу же выполняет функцию (`info`), и наша функция ничего не возвращает (внутри нет ключевого слова `return`), то не имеет никакого смысла записывать результат данной функции, так как он всегда в нашем случае будет равен `undefined`, поэтому создание констант `loggerMan` и `loggerWoman` делать не нужно.

Вот так вот просто. `call` – ничуть не страшнее `bind`. Вся разница лишь в том, что:

- `bind` **привязывает контекст и возвращает новую функцию** и мы можем вызвать её в любом месте;
- `call` **привязывает контекст и сразу же вызывает функцию**

Собственно, больше о `call` и сказать нечего.

Метод `apply`

Ты должно быть заметил, что о методе `bind` было сказано очень много. О методе `call` – уже куда меньше, потому что это одно и тоже с одним исключением. Будешь смеяться, но метод `apply` – это то же самое, что и `call`. И существует только **одно отличие**.

Если метод `call` (как, собственно, и `bind`) первым аргументом принимает **контекст**, а дальше через **запятую принимает аргументы для функции** (получается, что аргументов может быть разное количество), то метод `apply` принимает всего **2 аргумента**:

- Контекст (так же как `bind` и `call`);
- Массив параметров.

Т.е. если в `bind` и `call` мы можем **передавать параметры функции через запятую**, то в `apply` мы должны **передать их в массиве**. И в этом вся разница. Давай перепишем код с `call` на `apply`.

Было:

```
1 Logger.info.call(man, 'мужской');
2 Logger.info.call(woman, 'женский');
```

Стало:

```
1 Logger.info.apply(man, ['мужской']);
2 Logger.info.apply(woman, ['женский']);
```

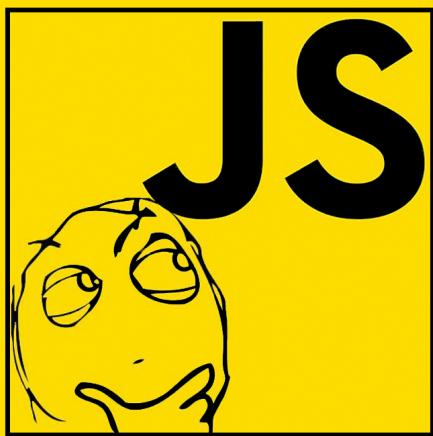
Вот собственно и вся разница. В случае с `call` мы передали аргумент `sex` для функции `info` как обычную строку. А в случае с `apply` мы обязаны передавать все аргументы в массиве.

Надеюсь, теперь ты поборол свой страх и понял, что всё, что связано с **контекстами**, конкретно, **данными методами** – совсем не страшно.

Замыкания

JavaScript для тупых

Замыкания



@javascriptstupid

Что же такое замыкание? Если говорить простыми словами, то это просто **функция в функции**. Вот и всё. Да-да. Вот так просто. **Это функция, которая находится внутри другой функции.**

Начнем сразу же с примеров, т.к. на них будет разобраться намного проще.

```
1 function hi() {  
2     return function() {  
3         console.log('Привет');  
4     }  
5 }
```

Такая вот незамысловатая конструкция: у нас имеется функция `hi`, которая возвращает другую функцию.

Получается, что если вызвать функцию `hi`, то может показаться, что ничего не произошло, так как в консоль не будет выведено слово `Привет`:

```
hi(); // в консоль ничего не выведется
```

Так как функция `hi` возвращает новую функцию, то нам нужно её вызвать, перед этим, для наглядности запишем её в константу и вызовем:

```
1 const sayHi = hi();
2
3 sayHi(); // выведет в консоль слово "Привет"
```

Только после этой манипуляции мы сможем визуально увидеть результат.

Итак, первым шагом мы разобрались, что функция `hi` в результате своего выполнения возвращает новую функцию. Но пока что совсем непонятно зачем это нужно.

Давай разбираться зачем это нужно.

В нашу функцию `hi` добавим параметр (аргумент), в него запишем имя человека, а в функцию, которая сейчас выводит нам слово `Привет` в консоль добавим это имя. В итоге получим такой код:

```
1 function hi(name) {
2     return function() {
3         console.log(`Привет, ${name}`);
4     }
5 }
```

Теперь заставим всё это работать:

```
1 const sayHi = hi('Вася');
2 sayHi(); // выведет в консоль: Привет, Вася
```

Стоит отметить, что имя мы передавали только в функцию `hi`, однако, та функция, которую возвращает функция `hi` так же имеет доступ к этому имени и спокойно может его использовать. На самом деле это и есть **замыкание. Возвращаемая функция замыкает в себе значение переменной `name`.**

Для большего понимания поправим немного код:

```
1 const sayHiVasya = hi('Вася');
2 const sayHiPetya = hi('Петя');
3 sayHiVasya(); // выведет в консоль: Привет, Вася
4 sayHiPetya(); // выведет в консоль: Привет, Петя
```

Как видишь, мы передаем имена только в нашу функцию `hi`, и эти имена **замыкаются в возвращаемых функциях.**

Создание ссылок для соц. сетей

Может быть еще не совсем понятно, поэтому давай сделаем более близкий к практике пример использующий **замыкания**.

Давай сделаем с тобой некий генератор ссылок для социальных сетей.

```
1 function createSocLink(socialNetwork) {
2     return function(nickname) {
3         return `https://${socialNetwork}/${nickname}`;
4     }
5 }
```

Итак, сначала разберём все в этом генераторе, а потом разберёмся как им пользоваться.

В функцию `createSocLink` мы передаём параметр `socialNetwork`, который должен представлять собой ссылку на какую-либо социальную сеть, т.е. должен иметь вид, к примеру:

- `vk.com`
- `instagram.com`
- `facebook.com`

Эта функция возвращает нам другую функцию, которая также ждёт параметра, указывающего на **никнейм пользователя** в указанной **социальной сети**. Т.е. когда мы будем вызывать ту функцию, которую нам вернёт функция `createSocLink` мы должны будем передать в неё параметр с указанием никнейма.

Итак, пробуем всё это запустить.

Так как функция `createSocLink` вернёт нам новую функцию, то запишем её в константу:

```
const createVkLink = createSocLink('vk.com');
```

Теперь в `createVkLink` у нас лежит функция, которая умеет делать ссылки для социальной сети **Вконтакте**. Теперь вызовем эту функцию и не забудем передать в неё аргумент, который должен указывать на **никнейм пользователя**:

```
1 const createVkLink = createSocLink('vk.com');
2
3 console.log(createVkLink('durov'));
4 // в консоль выведется: https://vk.com/durov
```

Как видишь, мы создали ссылку на страницу создателя Вконтакте Павла Дурова.

Теперь давай создадим ссылки с помощью генератора на другие соц. сети и ты увидишь, в чём прелесть замыканий:

```
1 const createVkLink = createSocLink('vk.com');
2 const createInstagramLink = createSocLink('instagram.com');
3 const createFacebookLink = createSocLink('facebook.com');
```

Мы получили **3 новых функции**:

- `createVkLink` – создаём ссылки на страницы пользователей во **Вконтакте**;
- `createInstagramLink` – создаём ссылки на страницы пользователей в **Instagram**;
- `createFacebookLink` – создаём ссылки на страницы пользователей в **Facebook**.

Попробуем их в работе:

```
1 console.log(createVkLink('durov'));
2 console.log(createVkLink('admin'));
3 console.log(createVkLink('vasya'));
4 console.log('*****');
5 console.log(createInstagramLink('durov'));
6 console.log(createInstagramLink('admin'));
7 console.log(createInstagramLink('vasya'));
8 console.log('*****');
9 console.log(createFacebookLink('durov'));
10 console.log(createFacebookLink('admin'));
11 console.log(createFacebookLink('vasya'));
```

Получаем результат:

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The output area displays a series of generated URLs:

```
https://vk.com/durov
https://vk.com/admin
https://vk.com/vasya
*****
https://instagram.com/durov
https://instagram.com/admin
https://instagram.com/vasya
*****
https://facebook.com/durov
https://facebook.com/admin
https://facebook.com/vasya
```

Ссылки прекрасно сгенерировались. Благодаря **замыканию**, мы получили список из 3-х генераторов ссылок:

```
1 const createVkLink = createSocLink('vk.com');
2 const createInstagramLink = createSocLink('instagram.com');
3 const createFacebookLink = createSocLink('facebook.com');
```

И каждый генератор теперь может генерировать сколько угодно ссылок, при этом, в него нужно передавать только **никнейм пользователя**, а сам адрес социальной сети будет браться из **замыкания**.

Домашнее задание

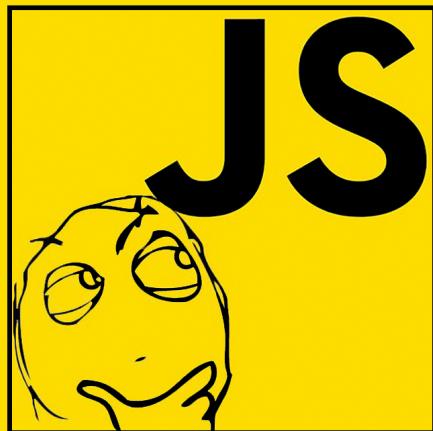
Создать функцию `calc` используя **замыкания**, которую можно использовать следующим образом:

```
1 const plus = calc(10);
2 plus(5); // должно вывести значение 15
3 plus(45); // должно вывести значение 55
```

Прототипы

JavaScript для тупых

Прототипы



@javascriptstupid

Чтобы лучше всё понять – сразу же создаём простой объект.

```
1 const cat = {  
2     name: 'Кот',  
3     weight: 3,  
4     meow: function() {  
5         console.log('meow');  
6     }  
7 };  
8  
9 console.log(cat);
```

Результат:

```
▼ Object {  
  ► meow: f ()  
  name: "Кот"  
  weight: 3  
  ► __proto__: Object
```

Попробуем вывести в консоль результат действия функции `meow`:

```
console.log(cat.meow()); // выведет строку: meow
```

Все логично и просто. Попробуем вызвать несуществующую функцию:

```
cat.woof();
```

Результат ожидаем:

✖ ► **Uncaught TypeError: cat.woof is not a function**
at main.js:11

JavaScript правильно подсказывает – `woof` не является функцией. Оно так и есть, ведь мы не определяли эту функцию внутри нашего объекта.

Но, для примера, давай попробуем вызвать ещё один метод, который мы не определяли:

```
cat.valueOf();
```

Результат:

```
> cat.valueOf()  
< ► {name: "Кот", weight: 3, meow: f}
```

Ошибки не произошло. И даже вывелись все данные о нашем объекте.

Попробуем еще:

```
cat.toString();
```

Результат:

```
> cat.toString();  
< "[object Object]"
```

Снова что-то вывелоось и снова никакой ошибки. Но как так?

Мы не определяли никакого метода `valueOf` внутри нашего объекта, ровно, как и не определяли метод `toString`. Магия вне Хогвартса? Нет. Все куда проще.

Еще раз посмотрим на первый пример:

```
▼ Object i  
► meow: f ()  
  name: "Кот"  
  weight: 3  
► __proto__: Object
```

Это наш объект. Показаны все **свойства и методы**, которые мы определяли: `meow` , `name` , `weight` .

Но что это за свойство `__proto__` ? Давай посмотрим, что лежит внутри него:

```
< ▶ {name: "Кот", weight: 3, meow: f, toString: f} ⓘ
  ► meow: f ()
    name: "Кот"
  ► toString: f ()
    weight: 3
  ▶ __proto__:
    ► constructor: f Object()
    ► hasOwnProperty: f hasOwnProperty()
    ► isPrototypeOf: f isPrototypeOf()
    ► propertyIsEnumerable: f propertyIsEnumerable()
    ► toLocaleString: f toLocaleString()
    ► toString: f toString()
    ► valueOf: f valueOf()
    ► __defineGetter__: f __defineGetter__()
    ► __defineSetter__: f __defineSetter__()
    ► __lookupGetter__: f __lookupGetter__()
    ► __lookupSetter__: f __lookupSetter__()
    ► get __proto__: f __proto__()
    ► set __proto__: f __proto__()
```

Очень много всего. Непонятно зачем, а главное – непонятно откуда.

Во всём этом списке, мы видим и те **методы**, которые мы вызывали: `valueOf` и `toString`.

Когда мы вызывали данные методы, **JavaScript** искал их сначала в пределах нашего объекта, а так как он не нашел их там, то пошёл искать их в свойство `__proto__`.

Как это работает и для чего нужно?

Давай сначала разберёмся с созданием **объекта**. Тот синтаксис, который мы использовали для определения нашего объекта `cat` является упрощенным:

```
1 const cat = {
2   name: 'Кот',
3   weight: 3,
4   meow: function() {
5     console.log('meow');
6   }
}
```

```
7 };
```

Мы можем определить это другим методом, тем, который более понятен для самого **JavaScript** и в который в любом случае, **JavaScript** приводит наш метод определения:

```
1 const cat = new Object({  
2     name: 'Кот',  
3     weight: 3,  
4     meow: function() {  
5         console.log('meow');  
6     }  
7});
```

Т.е. создаётся новый объект типа `Object` используя ключевое слово `new`. И внутрь этого `Object` мы передаём наш **объект**. В результате ничего абсолютно не меняется. Попробуем вывести объект `cat` в консоль:

```
console.log(cat);
```

Результат:

```
▼ {name: "Кот", weight: 3, meow: f} ⓘ  
  ► meow: f ()  
    name: "Кот"  
    weight: 3  
  ► __proto__: Object
```

Как видишь, мы поменяли метод инициализации нашего объекта, но абсолютно ничего не поменялось в итоговом результате.

Так как мы создаём все наши объекты используя эту конструкцию `new Object(...)`, то от этого самого `Object` к нашему объекту добавляются дополнительные свойства, к которым и относится то самое, непонятно откуда взявшееся, до текущего момента, свойство `__proto__`.

Получается, все объекты, которые мы создаем основываясь на базовом классе **JavaScript - Object**.

У класса `Object` имеется свойство `prototype`. Посмотрим, что там внутри:

```
console.log(Object.prototype);
```

Результат:

```
> Object.prototype
< ▷ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...} ⓘ
  ► constructor: f Object()
  ► hasOwnProperty: f hasOwnProperty()
  ► isPrototypeOf: f isPrototypeOf()
  ► propertyIsEnumerable: f propertyIsEnumerable()
  ► toLocaleString: f toLocaleString()
  ► toString: f toString()
  ► valueOf: f valueOf()
  ► __defineGetter__: f __defineGetter__()
  ► __defineSetter__: f __defineSetter__()
  ► __lookupGetter__: f __lookupGetter__()
  ► __lookupSetter__: f __lookupSetter__()
  ► get __proto__: f __proto__()
  ► set __proto__: f __proto__()
```

И мы видим, что внутри этого свойства лежат те же **методы**, которые добавились к нашему **объекту в свойство `__proto__`**.

Теперь, примерно объясню, для чего это нужно и как это можно использовать.

Давай в свойство `prototype` класса `Object` добавим какую-нибудь свою функцию. К примеру, функцию `woof`, которую мы пытались вызывать и у нашего объекта, но у нас всплывала ошибка:

```
1 Object.prototype.woof = function() {
```

```
2     console.log('woof');
3 }
```

Теперь, ничего не меняя в нашем объекте, т.е. он останется такого же вида:

```
1 const cat = new Object({
2     name: 'Кот',
3     weight: 3,
4     meow: function() {
5         console.log('meow');
6     }
7});
```

Попробуем вызвать метод `woof`:

```
console.log(cat.woof());
```

Результат:

```
> cat.woof();
```

```
woof
```

Как видишь, никакой ошибки и всё прекрасно отработало. Ещё раз посмотрим на наш объект в консоли:

```
console.log(cat);
```

Результат:

```
↳ ▶ {name: "Кот", weight: 3, meow: f} ⓘ  
▶ meow: f ()  
  name: "Кот"  
  weight: 3  
  ▶ __proto__:  
    ▶ woof: f ()  
    ▶ constructor: f Object()  
    ▶ hasOwnProperty: f hasOwnProperty()  
    ▶ isPrototypeOf: f isPrototypeOf()  
    ▶ propertyIsEnumerable: f propertyIsEnumerable()  
    ▶ toLocaleString: f toLocaleString()  
    ▶ toString: f toString()  
    ▶ valueOf: f valueOf()  
    ▶ __defineGetter__: f __defineGetter__()  
    ▶ __defineSetter__: f __defineSetter__()  
    ▶ __lookupGetter__: f __lookupGetter__()  
    ▶ __lookupSetter__: f __lookupSetter__()  
    ▶ get __proto__: f __proto__()  
    ▶ set __proto__: f __proto__()
```

Метод `woof`, который мы добавляли в свойство `prototype` объекта `Object` успешно передалось в свойство `__proto__` нашего объекта, поэтому и не произошло никакой ошибки.

Естественно, даже если мы будем использовать первоначальный синтаксис создания нашего объекта без использования `new Object(...)`, то всё равно всё будет работать ровно так же.

Надеюсь, ты уже понял, что с помощью прототипов мы получаем возможность расширять возможности наших объектов. К примеру, если оставить данный метод:

```
1  Object.prototype.woof = function() {  
2      console.log('woof');  
3  }
```

И создать несколько своих объектов, то для каждого из них в свойстве `__proto__` добавится метод `woof`. Т.е. написав этот метод один раз в базовом классе `Object` – мы можем использовать его в неограниченном количестве объектов созданных нами.

Object.create

У базового класса `Object` существует метод `create`. Он напрямую связан с прототипами, поэтому я решил рассказать и о нём.

Для того чтобы объяснить, как работает данный метод и что он делает, создадим **2 объекта**.

Первый объект будет представлять собой основу для второго. Называться он будет `employee`, что в переводе означает **работник**.

```
1 const employee = {  
2     name: 'Работник',  
3     position: 'Повар'  
4 };  
5  
6 console.log(employee);
```

Вывод в консоль нам даст:

```
▼ {name: "Работник", position: "Повар"} ⓘ  
  name: "Работник"  
  position: "Повар"  
▶ __proto__: Object
```

Для создания **второго объекта** нам понадобится метод `create` из класса `Object`:

```
const manager = Object.create(employee);
```

Что мы только что сделали? Мы создали новый объект , который должен описывать **работника**, являющимся **менеджером**.

`Object.create(employee)` говорит о том, что нужно создать новый объект, прототипом которого будет являться объект `employee` .

Давай выведем наш объект в консоль:

```
console.log(manager);
```

Результат:

```
▼ {} ⓘ  
► __proto__: Object
```

Объект пустой. И это неудивительно, мы же не задали ему ни одного свойства и метода. Но, что же у него теперь находится в свойстве `__proto__` ? Смотрим:

```
▼ {} ⓘ  
▼ __proto__:  
  name: "Работник"  
  position: "Повар"  
► __proto__: Object
```

А внутри него лежит объект `employee` , как мы и заказывали. И как видишь, у этого объекта, тоже есть свое свойство `__proto__` :

```
▼ {} ⓘ  
  ▼ __proto__:  
    name: "Работник"  
    position: "Повар"  
      ▼ __proto__:  
        ► constructor: f Object()  
        ► hasOwnProperty: f hasOwnProperty()  
        ► isPrototypeOf: f isPrototypeOf()  
        ► propertyIsEnumerable: f propertyIsEnumerable()  
        ► toLocaleString: f toLocaleString()  
        ► toString: f toString()  
        ► valueOf: f valueOf()  
        ► __defineGetter__: f __defineGetter__()  
        ► __defineSetter__: f __defineSetter__()  
        ► __lookupGetter__: f __lookupGetter__()  
        ► __lookupSetter__: f __lookupSetter__()  
        ► get __proto__: f __proto__()  
        ► set __proto__: f __proto__()
```

И как ты заметил это класс `Object`.

У нас получилась цепочка:

1. Создав объект `employee` обычным способом, мы получили объект, прототипом которого является базовый класс `Object`.
2. Мы создали объект `manager` с помощью `Object.create(employee)`, прототипом которого указали объект `employee`.

Теперь, имея объект `manager`, давай установим для него имя и должность:

```
1 manager.name = 'Сергей';  
2 manager.position = 'Менеджер';
```

Теперь еще раз посмотрим на наш объект:

```
▼ {name: "Сергей", position: "Менеджер"} ⓘ  
  name: "Сергей"  
  position: "Менеджер"  
▼ __proto__:  
  name: "Работник"  
  position: "Повар"  
► __proto__: Object
```

У нашего объекта появились свои свойства `name` и `position`, при этом одноименные свойства в прототипе (`employee`) никак не изменились.

В целом это всё что нужно знать о **прототипах**.

Если подытожить, то получается, что **прототип – это базовый объект другого объекта**. Этот базовый объект присутствует у других объектов и каким-то образом **расширяет** их возможности.

Кругом обман

На самом деле, все предыдущие материалы о **типах данных в JavaScript** были немного прикрыты страшной тайной. **На самом деле в JavaScript – нет строкового типа, числового, логического и т.д. В JavaScript – всё объекты.**

Это достаточно просто понять. К примеру, создадим обычную строку:

```
const str = 'строка';
```

Несмотря на то, что это обычная строка – у неё есть **методы**. К примеру:

```
console.log(str.toUpperCase()); // выведет: "СТРОКА"
```

Мы не определяли этот метод, но он существует. Он переводит всю строку к верхнему регистру.

Это происходит потому, что строковый тип, это на самом деле тоже объект, а именно `String`. Который в свою очередь основан на объекте `Object`.

Получается, что фактически написав такую вот конструкцию:

```
const str = 'строка';
```

Для **JavaScript** это то же самое что и:

```
const str = new String('строка');
```

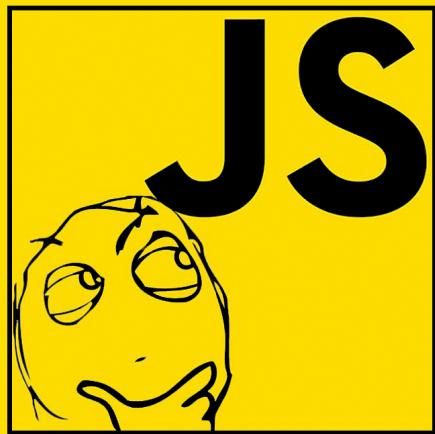
Строка `str` на самом деле является **объектом класса** `String`, а **прототипом объекта класса** `String` является **объект класса** `Object`.

Все в **JavaScript** создается на основе **класса** `Object`. Словом, он здесь главный.

Promises (Обещания)

JavaScript для тупых

Promises (Обещания)



@javascriptstupid

Хотелось бы начать с того, что `JavaScript` язык **синхронный**, т.е. весь код выполняется **последовательно**. И всё бы хорошо, но достаточно быстро в языке появилась потребность в дополнительных возможностях, а именно, понадобилась **асинхронность**. Для чего?

Все мы сидим в социальных сетях. Когда ты заходишь, например, в **Facebook**, то наверное, замечал, что сайт загружается за несколько секунд, а не **висит** по несколько минут. Все крупные сайты, как **Facebook** не смогли бы так быстро грузиться без **асинхронных операций**.

Ведь когда ты заходишь на **Facebook** – браузер отсылает очень много запросов на сервера соц.сети, чтобы получить какие-то данные. Например, фотографию твоего профиля, информацию из этого профиля, личные сообщения, списки друзей и т.п. На всё это нужно время, и если бы все эти запросы выполнялись друг за другом, т.е. каждый бы запрос ожидал завершения предыдущего – это было бы долго.

Благодаря **асинхронности**, все запросы отсылаются одновременно и весь остальной код не ждёт ответа от них, а **продолжает выполняться**. Когда же **асинхронная операция** заканчивает своё выполнение – отрабатывает какая-либо заранее подготовленная функция, которая, например, отобразит блок с твоими друзьями или все сообщения в определенном диалоге.

Какие асинхронные операции мы знаем?

Я уже писал об асинхронных методах в `JavaScript`, а конкретно: `setInterval` и `setTimeout`.

Это яркие представители **асинхронного исполнения кода**. Если не читал о них, то советую прочитать (делай тык):

- `setInterval`
- `setTimeout`

Эмуляция работы с сервером с помощью `setTimeout`

Итак, давай попробуем сэмюлировать работу с сервером. Сначала, сделаем это с помощью `setTimeout`, а затем с помощью `Promise` и, как итог, поймем в чем разница и рассмотрим все плюсы `Promise`.

Представим, что мы отсылаем запрос на сервер. Сервер собирает данные (на это уходит какое-то `n` время), потом сервер высыпает нам данные (это тоже занимает время).

Пошли к коду:

```
1 console.log('Отправляем запрос на сервер...');  
2  
3 setTimeout(function() {  
4     console.log('Сервер собирает данные...');  
5  
6     const data = {  
7         text: 'Данные с сервера'  
8     };  
9  
10    setTimeout(function() {
```

```
11     data.other = true;
12     console.log('Данные, которые предоставил сервер: ', data);
13 }, 2000);
14 }, 1500);
```

Итак, весь процесс нашей эмуляции:

1. Эмулируем отправку запроса
2. Создаём первый `setTimeout`, который отработает через `1.5` секунды. Этот таймаут будет эмулировать **сбор данных** и создаст объект `data` с этими данными.
3. Внутри первого `setTimeout` создаем второй. Он будет как-то дополнять/изменять объект `data` и как бы высыпать пользователю и потратит он на это `2` секунды.

Итог работы:

Отправляем запрос на сервер...

Сервер собирает данные...

Данные, которые предоставил сервер: { text: 'Данные с сервера', other: true }

Все прикольно. Отработало это ровно так как и ожидали за `3.5` секунды.

Вроде бы все прикольно, да не совсем. Мне лично, даже смотреть на такой код больно. `setTimeout` в `setTimeout`-е. Ведь если мы сейчас захотим ещё что-то эмулировать, то у нас будет ещё одна вложенность и все это будет напоминать **матрешку**. Разбираться в таких матрешках – не самое приятное удовольствие.

Поэтому, давай попробуем всё то же самое провернуть с помощью `Promise`.

Эмуляция работы с сервером с помощью `Promise`

Сначала создадим пустой `Promise`. Делается это так:

```
const promise = new Promise(function(resolve, reject) {});
```

Создается обещание с помощью класса `Promise`, поэтому используется ключевое слово `new`. В конструктор данного класса передаётся всего один аргумент – **callback-функция**, которая, в свою очередь, принимает в себя **2 аргумента**:

- `resolve` (переводится как **разрешить**)
- `reject` (переводится как **отклонить**)

На самом деле эти аргументы являются **функциями**. Благодаря этим 2-м функциям мы можем контролировать выполнение `Promise`. К примеру, внутри обещания у нас будут какие-то проверки. Если все проверки будут выполнены удачно, то мы вызовем функцию `resolve` и, в таком случае, `Promise` завершится удачно. А если же, какая-то проверка не будет пройдена, то мы сможем завершить работу `Promise` с помощью вызова функции `reject`.

Скорее всего ты пока ничего не понял. Не страшно, сейчас разберёмся со всем этим делом, не переживай.

Мы сделаем ту же самую эмуляцию работы с сервером с помощью `Promise`

Мы уже создали обещание, но оно ещё никак не функционирует. Поэтому, давай переносить функционал из кода написанного ранее.

```
1 console.log('Отправляем запрос на сервер...');  
2  
3 const promise = new Promise(function(resolve, reject) {  
4     setTimeout(function() {  
5         const data = {  
6             text: 'Данные с сервера'  
7         };  
8     }, 2000);  
9 });
```

Итак, внутри **callback-функции** нашего `Promise`, мы разместили код:

```
1 setTimeout(function() {  
2     const data = {
```

```
3     text: 'Данные с сервера'
4   };
5 }, 2000);
```

Это код нашего первого `setTimeout` из кода выше. Давай добавим в этот `setTimeout` вызов функции `resolve()`, так как мы хотим чтобы наш `Promise` выполнился без ошибок. В итоге получаем такой код:

```
1 const promise = new Promise(function(resolve, reject) {
2   setTimeout(function() {
3     console.log('Сервер собирает данные...');
4     const data = {
5       text: 'Данные с сервера'
6     };
7     resolve(); // успешное выполнение Promise
8   }, 1500);
9 });
10 );
```

Итак, как работать с этим всем дальше? У **callback-функции** `Promise`, как я и написал ранее существует две функции: `resolve`, `reject`.

Вызвав функцию `resolve` в коде выше, мы, как бы послали сигнал, что `Promise` успешно выполнился. Но как отловить этот сигнал? На самом деле в этом нет ничего сложного.

В константу `promise` мы записали наш `Promise`, поэтому мы можем работать с ней следующим образом:

```
1 promise.then(function() {
2   console.log('Успешное выполнение Promise');
3 });
```

Но, в целом, в этом случае (как и во многих других) лучше использовать **стрелочную функцию** в качестве `callback`:

```
promise.then(() => console.log('Успешное выполнение Promise'));
```

У `Promise` существует метод `then`, который ожидает, что в него ты передашь **callback-функцию**, она выполнится только в тот момент, когда внутри самого `Promise` мы вызовем функцию `resolve`, означающая успешное выполнение `promise`.

Итог работы нашего кода:

Отправляем запрос на сервер...

Сервер собирает данные...

Успешное выполнение Promise

Итак, первую часть мы реализовали с помощью `Promise`. Пока что непонятно, чем же `Promise` лучше и в чём их профит. Но давай продолжим перетаскивать код дальше.

На самом деле, третьим сообщением, в соответствии с первой реализацией эмуляции должен выводится текст: «**Данные, которые предоставил сервер...**» и дополнительного должен выводится сам объект `data`, а не «**Успешное выполнение Promise**». Давай это поправим, поэтому вернемся к этой строке:

```
promise.then(() => console.log('Успешное выполнение Promise'));
```

Итак, здесь мы должны заменить текст и вывести объект `data`. Но вот в чём проблема – здесь, в этой **callback-функции** у нас нет никакого объекта `data`, следовательно, вывести мы его не можем. Чтобы решить данный вопрос и получить доступ к объекту `data` нужно всего лишь в метод `resolve`, который мы вызываем

В `Promise` передать наш объект `data`. Вернёмся к нашему коду и поправим вызов `resolve`:

```
1 const promise = new Promise(function(resolve, reject) {  
2   setTimeout(function() {  
3     console.log('Сервер собирает данные...');  
4     const data = {  
5       text: 'Данные с сервера'  
6     };  
7  
8     resolve(data); // передаём data  
9   }, 1500);  
10});
```

Теперь в функцию `resolve` мы передаём наш объект `data`. Что же это нам даёт? А даёт это нам возможность получить этот объект в методе `then`. И вот как это делается.

Вот это:

```
promise.then(() => console.log('Успешное выполнение Promise'));
```

Меняем на:

```
1 promise.then(data =>  
2   console.log('Данные, которые предоставил сервер: ', data)  
3 );
```

Как видишь, теперь у нас **стрелочная функция** имеет аргумент `data` – и в этот аргумент и попадает то, что мы передаём внутрь функции `resolve` при её вызове.

Как итог, получаем:

Отправляем запрос на сервер...

Сервер собирает данные...

Данные, которые предоставил сервер: { text: 'Данные с сервера' }

И все бы хорошо, да вот только объект в итоге имеет совсем не тот итоговый вид, как в первом случае.

На данный момент мы перенесли только один `setTimeout`, а у нас их было два.

Во втором `setTimeout` мы добавляли дополнительно поле `other` со значением `true` к нашему объекту `data`.

Получается, что мы пропустили момент **модификации объекта**. Давай восполним данную потерю.

Нам нужно в какой-то момент модифицировать объект `data` и добавить ему свойство `other`. Когда же нам это сделать? Сделать нам это нужно здесь:

```
1 promise.then(data =>
2   console.log('Данные, которые предоставил сервер: ', data)
3 );
```

Вместо того, чтобы просто выполнить `console.log`, нам нужно изменить объект `data`, который приходит к нам из `resolve` выполненного в `Promise`. Более того, нам нужно выполнить это только через 2 секунды, а это значит, что нам нужно добавить наш `setTimeout`.

Давай попробуем исправить наш код:

```
1 promise.then(data =>
2   setTimeout(function() {
3     data.other = true;
4     console.log('Данные, которые предоставил сервер: ', \
5       data);
6   }, 2000)
```

```
7 );
```

Итог:

Отправляем запрос на сервер...

Сервер собирает данные...

Данные, которые предоставил сервер: { text: 'Данные с сервера', other: true }

И вот, вроде бы уже можно вскрикнуть «Ура». Fail. На самом деле, мы схалтурили.

Первый `setTimeout`, мы реализовали внутри `Promise`, что обеспечило нам контроль над происходящим: можем вызвать `resolve` для того, чтобы указать на **успешное выполнение** и `reject` – на **выполнение с ошибкой**.

Сейчас же, наш второй `setTimeout` не имеет такой возможности. А всё потому, что мы не использовали `Promise`. Давай используем его и поправим наш имеющийся код. Для того, чтобы добавить `Promise`, нам нужно, чтобы **callback-функция**, которую мы определяем внутри `then` – возвращает нам новый `Promise`. Поэтому давай снова изменим **callback-функцию** в `then`:

```
promise.then(data => new Promise(function(resolve, reject) {}));
```

Теперь мы сделали так, что `then` вернёт нам новый `Promise`. Пока что он ничего не выполняет, поэтому давай добавим наш `setTimeout` в тело **callback-функции** нашего нового `Promise`:

```
1 promise.then(data => new Promise(function(resolve, reject) {
2     setTimeout(function() {
3         data.other = true;
4
5         // не забываем выполнять resolve(data)
6         resolve(data);
7     }, 2000)
8 }));
```

Этим этапом мы изменили наш объект `data`. И с помощью `resolve (data)` мы сообщили, что наш новый (второй) `Promise` выполнился успешно и передал наш объект `data` дальше.

Но мы ещё не выводим сообщение о том, что сервер предоставил нам какие-то данные. Давай поправим и это. Так как у нас из `then` возвращается новый `Promise`, то это означает, что мы можем использовать тот же метод `then` к этому обещанию и как-то отреагировать на новый вызов `resolve (data)`:

```
1 promise.then(data =>
2   new Promise(function(resolve, reject) {
3     setTimeout(function() {
4       data.other = true;
5
6       // не забываем выполнять resolve(data)
7       resolve(data);
8     }, 2000)
9   })
10 ).then(data =>
11   console.log('Данные, которые предоставил сервер: ', data)
12 );
```

И вот теперь у нас все выполняется абсолютно так, как нужно.

Итак, весь наш код выглядит следующим образом:

```

1  console.log('Отправляем запрос на сервер...');
2
3  const promise = new Promise(executor: function(resolve, reject) {
4    setTimeout(handler: function() {
5      console.log('Сервер собирает данные...');
6      const data = {
7        text: 'Данные с сервера'
8      };
9
10     resolve(data);
11   }, timeout: 2000);
12 });
13
14 promise.then(data =>
15   new Promise(executor: function(resolve, reject) {
16     setTimeout(handler: function() {
17       data.other = true;
18
19       resolve(data);
20     }, timeout: 2000);
21   })
22 ).then(data =>
23   console.log('Данные, которые предоставил сервер: ', data)
24 );

```

По количеству строк, относительно изначальной реализации – кода прибавилось. Но если задуматься, то в первой нашей реализации мы никак *не управляли состоянием выполнения и не могли на него повлиять*.

С `Promise` же, мы **имеем контроль над выполнением нашего кода**. Если код успешно выполнился, то мы выполняем функцию `resolve` и обработчик `then` сразу же отлавливает это и выполняет заданные нами действия. Это же круто? Безусловно.

Но, мы пока что не затронули метод `reject`. Поэтому, давай поговорим и о нём.

Метод `reject`

Говорим и говорим о `resolve`, а `reject` как будто, вообще никто и звать никак.

На самом деле метод `reject` не менее полезный, но служит он для той цели, чтобы сообщить о том, что наш `Promise` должен завершиться ошибкой.

Ты уже знаешь, что обработчиком выполнения функции `resolve` служит метод `then`.

У функции `reject`, свой обработчик – `catch`.

Если посмотреть ещё раз на последний пример с кодом, то можно заметить, что `Promise` создаёт цепочку вызовов методов `then`:

```
1 const promise = new Promise(...);
2
3 promise.then(
4   ... код ...
5 ).then(
6   ... код ...
7 )
```

Так вот, во всей этой цепочке, методу `catch` самое место – практически в самом её конце (почему практически – узнаешь дальше):

```
1 const promise = new Promise(...);
2
3 promise.then(
4   ... код ...
5 ).then(
6   ... код ...
7 ).catch(
8   ...обработка ошибки...
9 )
```

Внутри этого `catch` мы можем каким-то образом обработать ошибку и, как и с функцией `resolve`, мы можем передать эту самую ошибку в качестве аргумента функции `reject(error)`.

Для примера, я поменяю в нашем коде один из `resolve` на `reject` и передам в качестве аргумента ошибку:

```
1 const promise = new Promise(function(resolve, reject) {
2   setTimeout(function() {
3     console.log('Сервер собирает данные...');
4     const data = {
5       text: 'Данные с сервера'
6     };
7
8     reject(new Error('Ошибка сбора данных')); // \
9       // передаем ошибку
10    }, 1500);
11  });
12
13 promise.then(data =>
14   new Promise(function(resolve, reject) {
15     setTimeout(function() {
16       data.other = true;
17
18       // не забываем выполнять resolve(data)
19       resolve(data);
20     }, 2000)
21   })
22 ).then(data =>
23   console.log('Данные, которые предоставил сервер: ', data)
24 ).catch(err => console.error(err));
```

Все что мы сделали – это вызвали `reject` и навесили обработчик `catch`. И теперь, если запустить наш код, мы получим ошибку:

```
Отправляем запрос на сервер...
Сервер собирает данные...
Error: Ошибка сбора данных
at Timeout._onTimeout (/Users/... /test.js:10:12)
at listOnTimeout (internal/timers.js:531:17)
at processTimers (internal/timers.js:475:7)
```

Метод `finally`

Кроме методов `then` и `catch`, существует еще один метод – `finally`.

Метод `finally` выполняется всегда, вне зависимости от того вызвали мы внутри обещания `resolve` или `reject`.

Этот метод мы размещаем в самом конце цепочки и итоговый код у нас получается таким:

```
1 const promise = new Promise(function(resolve, reject) {
2     setTimeout(function() {
3         console.log('Сервер собирает данные...');
4         const data = {
5             text: 'Данные с сервера'
6         };
7
8         resolve(data);
9
10        // генерируем ошибку
11        // reject(new Error('Ошибка сбора данных'));
12    }, 1500);
13 });
14
15 promise.then(data =>
16     new Promise(function(resolve, reject) {
17         setTimeout(function() {
18             data.other = true;
19
20             // не забываем выполнять resolve(data)
21             resolve(data);
22         }, 2000)
23     })
24 ).then(data =>
25     console.log('Данные, которые предоставил сервер: ', data)
26 ).catch(err =>
27     console.error(err)
28 ).finally(() =>
29     console.log('Работа с сервером завершена')
30 );
```

`finally` действительно не важно, будет ошибка:

Отправляем запрос на сервер...
Сервер собирает данные...
Работа с сервером завершена
Error: Ошибка сбора данных

или ее не будет:

Отправляем запрос на сервер...

Сервер собирает данные...

Работа с сервером завершена

Он будет выполняться всегда.

Домашнее задание

Итак, домашнее задание придумать не просто. Поэтому, я оставлю листинг кода в онлайн-редакторе, попробуй самостоятельно поиграться с кодом и разобраться. Ну, и, конечно же, если совсем будет много вопросов – всегда можешь писать в наш чат ([ссылка](#) на него в [группе](#), в закрепленном сообщении).

[Ссылка на код](#)